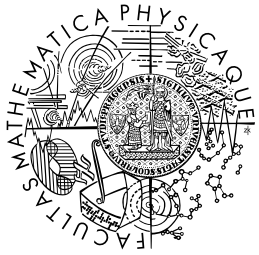

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Waltl

BRDF Editor

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	RNDr. Josef Pelikán
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor, RNDr. Josef Pelikán, for guiding me through this task with useful comments and ideas about the developed program and more importantly his patience and willingness to answer all my questions and helping with solving any problems.

Title: BRDF Editor

Author: Jan Waltl

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: The goal of this thesis is to create a working environment for the development and testing of bidirectional reflectance functions (BRDFs). The result of our work is a graphical application that offers tools to write these BRDFs, see how they behave on dynamic 2D graphs and in simple scenes. To achieve this, we created a general framework for physically based rendering algorithms. With the help of accelerating in hardware, in particular graphics cards(GPUs), we use OpenCL API to boost performance and allow interactive work with the developed functions. As part of the work, we implemented the path tracing algorithm capable of rendering realistic-looking scenes with indirect lighting from area lights and an environment light. The used algorithm uses importance sampling to greatly improve convergence speed and allows writing these custom sampling strategies for the written BRDFs and seeing how they match the BRDF, thus testing their effectiveness.

Keywords: BRDF OpenCL photo-realistic rendering path tracing GPU

Contents

Introduction

The goal of computer graphics is creating images and animations using computers and today is widely used for entertainment purposes such as movies and video games as well as for general visualization of collected data including medical diagnostics.

One of its goals is to create photo-realistic images that mimic the real world around us. The most straightforward way to do so would be to do it exactly like a camera captures a scene in the real world. Light can be modelled as a stream of photons emitted from light sources, travelling through the scene in straight lines - rays, interacting with objects and eventually hitting the viewer. Which might be a photographic paper, a digital light sensor in a camera, or a human eye. This idea is the core concept for ray tracing and path tracing algorithms. 1 mentions that only those photons that do reach the viewer contribute to the final image and so it is better to reverse the process. Instead of tracing the photons from the lights to the viewer we could shoot virtual photons from a place on the image, through the camera and trace their paths inside the scene. Some might hit a light source and if that happens a part of the light's energy is transferred to the original place on the image. This does not ensure that all photons will reach a light because there simply might not be a path to any of them, but then that part of the image will be black and the photon still transferred *information*.

The interaction of light with objects depends on their properties and so an object has a material that describes this interaction. In computer graphics, one way to approximate these materials is to use a bidirectional reflectance function(BRDF) which governs the surface interaction with the light. It is these materials that have a great impact on realism in the computer-generated images and so having proper tools to create and test these reflectance functions is important. The goal of this thesis is to create a tool which does exactly that and given the constant growth in computational power, more complex scenes can be used than in previous tools while still keeping the process interactive.

Text organization

The thesis can be partitioned into a theoretical and practical parts. In the first chapter, we introduce the necessary physical concepts of light. Then, over the next 5 chapters, the path tracing algorithm as stated in [2],[3],[4] is derived. First, its basic form in chapter 2, followed by explanation of light interactions with the scene. The fourth chapter introduces a few well-known reflectance functions. Chapter five expands on the brief introduction of Monte Carlo integration methods from chapter 2 and describes how are these methods implemented in computers using pseudo-random numbers. The sixth chapter revisits the path tracing algorithm to explain techniques which can greatly increase its speed of convergence through smarter use of Monte Carlo integration.

The second part of this thesis deals with the development of the editor itself. First, in chapter 8, we give an overview of existing programs, how could they be improved and what does our program do differently. Next chapter discusses various technologies that can be used for the development and achieving much-needed

performance. After deciding on the technologies, the tenth chapter describes the general architecture and features of the developed editor, with emphasis on the decisions one would have to make for developing their application including some technical ones.

The last, eleventh, chapter reflects on the created program and possible future improvements. Part of this text is also the user guide located at the end.

1. Light

This chapter will first concisely define and explain the physical properties of light that are necessary for building a physically-based renderer. Then a definition for the bidirectional reflectance distribution function (BRDF) will be given. The chapter will conclude with a definition of the core concept of light transport - the rendering equation [3].

1.1 Physical Quantities

Radiant flux is defined as radiant energy measured per given unit of time

$$\Phi = \frac{\partial Q}{\partial t}[W]. \tag{1.1}$$

$Q \ldots$ is radiant energy [J].

$t \ldots$ is time [s].

Irradiance(Radiant exitance) at the point x is the radiant flux received by or emitted from a given surface per unit area.

$$E(x) = \frac{\partial \Phi}{\partial A}\left[\frac{W}{m^2}\right] \tag{1.2}$$

$\Phi \ldots$ is radiant flux [W].

$A \ldots$ is area $[m^2]$.

Radiance at the point x is defined as the radiant flux received by or emitted from a given surface per unit projected area along a direction ω and per a unit solid angle around ω.

$$L(x,\omega) = \frac{\partial \Phi^2}{\partial A \cos(\theta) \partial \omega}\left[\frac{W}{m^2.sr}\right] \tag{1.3}$$

$\Phi \ldots$ is radiant flux [W].

$A \ldots$ is area $[m^2]$.

$\theta \ldots$ is the angle between the direction ω and the normal of the area A.

We will differentiate between L_i and L_o for the incoming and outgoing radiance, respectively. That is $Ł_i(x,\omega) = L(x,\omega)$, $Ł_o(x,\omega) = L(x,-\omega)$ which ensures that ω always points away from the surface. It can be shown, that for non-scattering volume between two points x, y the incoming radiance at the point x from the direction of $\omega = y - x$ is equal to the outgoing radiance from the point y in direction ω All these properties of light are depicted in fig. 1.1.

1.2 Bidirectional Reflectance Distribution Function - BRDF

The BRDF is defined as a ratio of radiance reflected from the surface point x along the outgoing direction w_o to the irradiance incoming from the incident

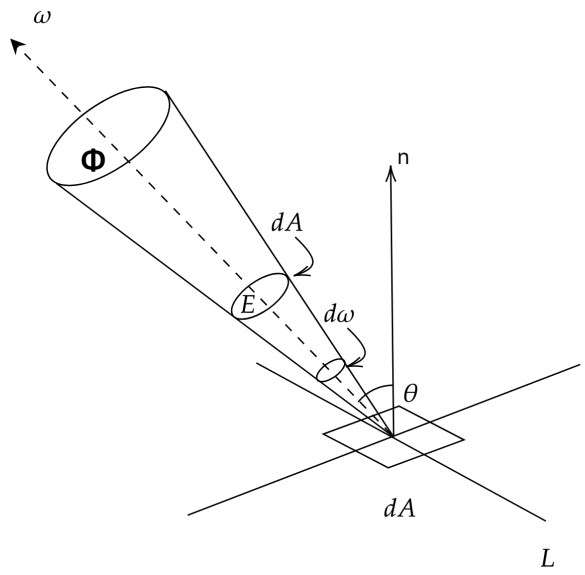

Figure 1.1: An example of all three physical quantities. The radiant flux Φ is emitted from a circular source. This flux is then measured through the unit area obtaining the irradiance E. Taking into account the angle between a surface gives the radiance L.

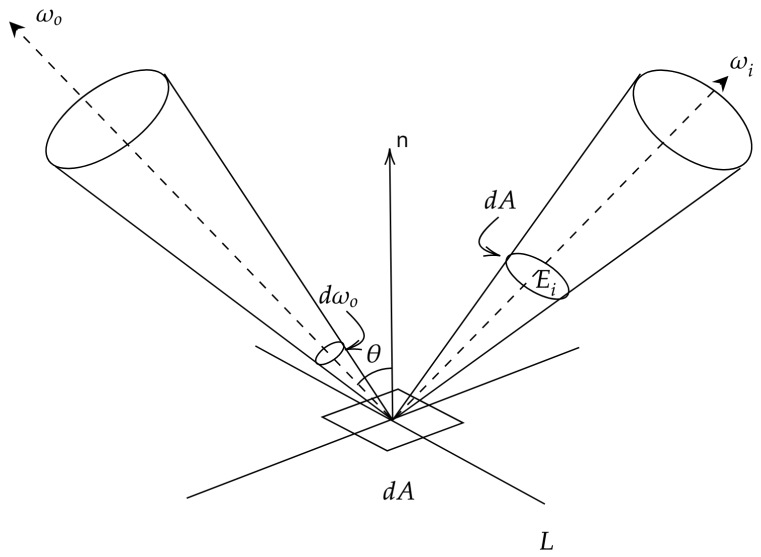

Figure 1.2: The definition of the reflectance function.

direction ω_i

$$f(x, \omega_i \to \omega_o, \lambda) = \frac{\partial L(x, \omega_o)}{\partial E(\omega_i)} = \frac{\partial L(x, \omega_o)}{L(x, \omega_i) \cos\theta \partial \omega_i}. \tag{1.4}$$

The second equality follows from an observation that $L(x, \omega_i) = \frac{\partial E(x)}{\cos\theta \partial \omega_i}$. The definition is also shown in fig. 1.2. Note that properties of materials may depend on the wavelength of light and this function does to. We will drop the extra λ argument for clarity and assume that it is always there.

Physically correct BRDF should satisfy additional constraints, in particular

1. Energy conservation: $\int_S f(x, \omega_i, \omega_o) d\omega_o \leq 1 \quad \forall \omega_i$ - i.e that a surface does not reflect more energy than received. This does not prohibit emissive objects because the BRDF only accounts for the reflected right.

2. Reciprocity: $f(x, \omega_i, \omega_o) = f(x, \omega_o, \omega_i)$.[Explain why]

3. Positivity: $f(x, \omega_i, \omega_o) \geq 0$.

First two are stated e.g. in [5], the last one should be trivial from definition as there is no "*negative*" light. Note that cases of the integral in the first constraint being strictly smaller than one are correct and quite common as usually at least some portion of the light is either absorbed or transmitted. Thus the BRDF is strictly speaking not a probability distribution as the name might suggest.

Later, fig. 9.3 shows an example of two distribution functions together with rendered images. chapter 4 is dedicated to a more in-depth description of the BRDF and contains proper definitions and formulas for a few well-known models.

The BRDF only accounts for reflected light and there is also a complementary bidirectional transmittance distribution function that is defined in the same way as BRDF but for transmitted light. Together they can be combined into bidirectional scattering distribution function. All three functions assume that the light enters and exits the surface at the same point. This is not true for many materials, in particular, this effect is necessary to realistically model e.g. wax and human skin, it can be modelled using a bidirectional scattering-surface reflectance function

1.3 Light transport equation

The light transport equation (LTE), or the rendering equation in [3], is the equation that tries to solve the propagation of light through a scene, radiance to be precise. A path tracing algorithm that will be presented later is an attempt to numerically solve the LTE. The equation can be formulated in more than one way. The first one can be obtained by straight-forward integration of the definition of the BRDF as shown in eq. (1.8) and was described in [2]. This form should be very intuitive - all incoming light is gathered from every direction $omega_i$, the reflectance function then exactly specifies how much should be reflected into a particular ω direction.

$$f(x, \omega_i \to \omega_o) = \frac{\partial L_o(x, \omega_o)}{\partial L_i(x, \omega_i) \cos\theta \partial\omega_i} \tag{1.5}$$

$$\partial L_o(x, \omega_o) = f(x, \omega_i \to \omega_o) \partial L_i(x, \omega_i) \cos\theta \partial\omega_i \tag{1.6}$$

$$L_o(x, \omega_o) = \int_{H^2} L_i(x, \omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) d\omega_i \tag{1.7}$$

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{H^2} L_i(x, \omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) d\omega_i \tag{1.8}$$

Few notes about this formulation. First, we only integrate over top hemisphere because that is where BRDF is properly defined. The BRDF is still dependent on the wavelength which is omitted and will be further explained in the next chapter. The term L_e represents emitted radiance from a surface of an object. Without this constant factor we would get the trivial solution $L \equiv 0$. It also corresponds to an idea of lights. The recurrent part is present through $L_i(x, \omega_i)$ that can be evaluated using the same equation with knowing that $L_i(x, \omega_i) = L_o(y, -\omega_i)$ for a point y which is closest unobstructed point from x in the direction of ω_i. This makes the y dependent on the scene.

The rest of the theoretical part will revolve around describing known methods for solving this execution at least numerically.

1.3.1 Area formulation

Evaluating the integral in eq. (1.8) would require evaluation for all incoming directions but this might not be really necessary. Another strategy is to not integrate over solid angle, but surface areas located in the scene. That is a valid approach since L_e is non-zero only on a surface of an object. I.e. radiance can only be *created* at emission. By projection arbitrary differential area dA to a unit sphere's surface it can be seen that relationship between dA and $d\omega$ is given by

eq. (1.9). With this knowledge eq. (1.8) can be rewritten as eq. (1.10). Evaluating this integral means going over all points of all surfaces in the scene. This form, albeit with slightly different notation, is introduced in [3]. It contains a new term $V(x, y)$ - *visibility* i.e. whether x is directly visible from y. This term was not present in the earlier formulation because it was implicitly contained in $L_i(x, \omega_i)$, as was mentioned earlier, which returns the radiance from the closest point. In this formulatio,m y might not be that point and ω_i is calculated as $\omega_i = \frac{y-x}{||y-x||}$.

$$
\begin{aligned}
d\omega &= \frac{\cos\gamma}{r^2} = \frac{-n_A.\omega}{} \\
\gamma &\dots \text{Angle between area's normal and } \omega \\
r &\dots \text{Distance from area's center to the surface's point.} \\
n_a &\dots \text{Area's surface normal.}
\end{aligned} \tag{1.9}
$$

$$
L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{A_y \in \mathbb{A}} L_i(x, \omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) \frac{-n_A.\omega_i}{||y - x||^2} V(x, y) dA_y \tag{1.10}
$$

1.4 Camera

Assuming that the radiance can be estimated at arbitrary point in the scene our goal is still to generate an image. Drawing from the real-world examples, the simplest model is probably a pinhole camera schematically depicted fig. 1.3. It does not contain any lenses and we will assume it has an infinitely smaller aperture and the sensor's sensitivity over all pixels is constant. Then, all the rays which hit its receptive field pass through a single point in space. We can construct a virtual camera in the scene using this knowledge and write a simple algorithm that can generate rays. Our final image is then computed simply by calculating arriving radiance from the scene to individual pixels.

Real, modern cameras are much more complex, they have non-zero size aperture, usually, more than one lens, and the receptive field is not uniformly sensitive to incoming radiance. This means that our simple camera cannot reproduce many real-world effect e.g. vignetting, chromatic aberrations, and depth of focus. But none of them are particularly important to our application.

1.4.1 Limitations

Although LTE is an attempt to reproduce physical behaviour of light it has its limitations. It does not consider the light transmitted through an object nor any form of sub-surface scattering even though effects are important for many real-world scenes in computer graphics. That said it can be generalized to cover these phenomena [5]. Neither it does take into account any wave-like properties of light and the presented versions do not contain a time term and thus cannot reproduce dynamic scenes. Another real-world phenomenon is volume-scattering which breaks the radiance transport invariance that we rely on. In this case, L_o will frequently not be equal to L_i in between two points in space. So, LTE cannot reproduce smoke, clouds or even sky. None of these are, again, in interest to us and so are not covered.

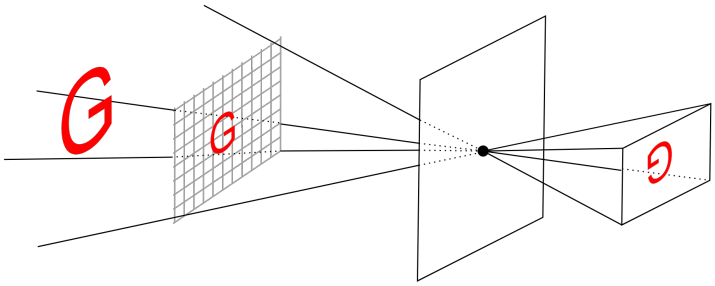

Figure 1.3: The pinhole camera works by separating the scene we want to capture from a receptive field using a plane with a very small hole in it. This generates mirrored, upside-down picture of the scene. Reflecting this receptive field around the plane generates a virtual receptive field and this model is in computer graphics commonly known as a perspective camera.

2. Solving Light Transport Equation

This chapter presents various developed strategies that can be used to solve or at least approximate a solution to the LTE and presents the first version of the path tracer algorithm that solves the rendering equation. All solutions are based on the following idea.

2.1 LTE as operator

[3] in the paper on the rendering equation exploits the recurrent nature of eq. (1.8) by defining an operator R like in eq. (2.1), which then, in turn, enables the LTE solution to be stated explicitly.

$$R[L](x,\omega_o) := \int_{H^2} L_i(x,\omega_i) f(p,\omega_i \to \omega_o)(\omega_i.n) d\omega_i \tag{2.1a}$$

$$L = L_e + R[L] \tag{2.1b}$$

$$L_e = (I - R)[L]$$

$$L = (I - R)^{-1}[L_e]$$

$$L = \sum_{n=0}^{\infty} R^n[L_e] = L_e + R[L_e] + R[R[L]] + R[R[R[L]]]... = \tag{2.1c}$$

This approach from [3] (and more thoroughly in [2]) expressed in eq. (2.1) allows us to rewrite the LTE in much more compact form. The last equality can be seen as an analogous solution to the geometric series. [3] notes on the equality between eq. (2.1b) and eq. (2.1c): "*A condition for the convergence of the infinite series is that the spectral radius of the operator* $\boldsymbol{T}$ *be less than one. (Which is met in the case of interest to us). A physical interpretation of the Neumann expansion is appealing. It gives the final intensity of radiation transfer between points x and* $\boldsymbol{y}$ *as the sum of a direct term, a once scattered term, a twice scattered term, etc.*". Citation has been adapted(bold) to our notation. The last sentence is the core idea for a path tracer. Since $R[L_e]$ is an ordinary integral without any recursion if it can be evaluated for arbitrary L_e, then the LTE can be approximated also with an arbitrary precision simply by repeatedly applying the R operator. And indeed, the next section presents a method that allows to at least estimate this integral and it is called *Monte Carlo integration*.

2.2 Monte Carlo integration

Monte Carlo integration is a numerical method that can be used to estimate a definite integral using random numbers. The technique is well-established and e.g. [6] can be consulted for a more detailed explanation of numerical integration in general and also the Monte Carlo methods. Consider this integral

$$A = \int_{\Omega \subseteq \mathbb{R}^N} f(x) dx, \tag{2.2}$$

then a Monte Carlo estimate for this integral is given as

$$\frac{C}{N}\sum_{i=0}^{N} f(\ddot{x}_i)$$

, where C is the volume of the domain Ω ($C = \int_{\Omega \subseteq \mathbb{R}^{\mathbb{N}}} 1dx$) and $\ddot{x}_i$ are uniformly distributed random numbers in Ω. [6] furthermore states that this estimate converges to the true value, meaning

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=0}^{N} f(\ddot{x}_i) = A$$

. The book gives a derivation of well-known convergence rate order of $O(\frac{1}{\sqrt{N}})$. The important thing is that this rate is independent of dimensionality of the integral which makes it viable technique to evaluate high-order $R^n[L_e]$ terms. Furthermore, all that is needed to estimate an integral is ability to generate random numbers and evaluate $f(x)$ for them.

2.3 LTE solution

In the section 1.3 two forms of the LTE were introduced:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{H^2} L_i(x,\omega_i) f(x,\omega_i \to \omega_o)(\omega_i.n)d\omega_i,$$
$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int\limits_{A_y \in \mathbb{A}} L_i(x,\omega_i) f(x,\omega_i \to \omega_o)(\omega_i.n)\frac{-n_A.\omega_i}{||y-x||^2}V(x,y)dA_y.$$

Now, we can use the observations from eq. (2.1) to estimate terms of eq. (2.1c):

$$R[L_e](x,\omega_o) = \frac{2\pi}{N}\sum_{i=0}^{N} L_e(x,\ddot{\omega}_i) f(x,\ddot{\omega}_i \to \omega_o)(\ddot{\omega}_i.n), \tag{2.3a}$$

$$R[L_e](x,\omega_o) = \frac{Area(\mathbb{A})}{N}\sum_{i=0}^{N} L_e(x,\ddot{\omega}_i) f(x,\ddot{\omega}_i \to \omega_o)(\ddot{\omega}_i.n)\frac{-n_A.\omega_i}{||y-x||^2}V(x,y), \tag{2.3b}$$

$$R^{k+1}[L_e] = R[R^k[L_e]]. \tag{2.3c}$$

In order to evaluate the integral we only need to pick a random directions or points in the scene. Note that $L_e(x,\omega_i)$ is not a recursive function and instead is property of the scene. Repeatedly applying this approach (with another sets of random variables) we can indeed estimate the LTE using eq. (2.1c) with arbitrary precision. We already stated that this approach is only an estimate with error rate proportional to $\frac{1}{\sqrt{N}}$ which might, and often will, require many samples to obtain a close-enough estimate.

2.3.1 Russian roulette

Even though one can now estimate any $R^k[L_e]$ term, we still cannot evaluate the infinite sum using a finite algorithm. Only evaluating first K terms for any fixed K will introduce systematic bias to the estimation. For the unbiased result we can at least try to estimate the sum itself. The tool that can be used for that is called Russian roulette and for this purpose was presented e.g. in [4] and similarly in [2]. First, one can notice that if any term in the sum is 0 then all subsequent terms will also be 0. This is true because the R operator does not contain any additive term. Let $p \in [0, 1]$ be a random number, $q \in (0, 1)$ fixed and define $R'[L_e]$ as

$$R'[L_e] = \begin{cases} \frac{R[L_e]}{q} & \text{if } p \leq q \\ 0 & \text{if } p > q \end{cases}$$

. Now, the observation is

$$E[R'[L_e](.,.) = q\frac{R[L_e]}{q} + (1-q)0 = R[L_e].$$

This means that the sum can be estimated with R' operator instead. Furthermore, any time the value after application of R' becomes zero the rest of the sum will be too. In theory, this algorithm is still not finite but the probability that we have to estimate the k-th term is q^k - an exponential falloff. In practice, there will be a maximum allowed number due to GPU implementation.

The value of q can change between the terms and [2] sets this value proportional to the BRDF inside the R operator, albeit it is used for terminating paths in bidirectional path tracer.

2.4 First path tracing algorithm

This approach gave us a first version of a path tracing algorithm that is presented in section 2.4. It uses solid angle formulation and only $N = 1$ samples to estimate the incoming radiance. That is true for all path tracing algorithms because it allows non-branching implementation. We can follow a path through the scene instead of branching trees emitting from the camera. We did not follow the path from a light to the camera as the sum does by starting at L_e, the reasoning for that was discussed in the introduction. This algorithm can be derived from a *path formulation* of the LTE and using the Russian roulette as was done in [4],[2], but we did not mention this form explicitly. The `througput` value holds all terms inside $R^k[L_o]$. Note that the `througput` is used[1] as the value of q from the previous section. Last notable thing is that for simpler implementation the objects are partitioned into two categories - *ordinary objects* (from now on referred to as objects) and *lights*. Both can be intersected, but only objects have materials, generate reflected rays and have $L_e(x, \omega_o) = 0$. When the ray hits a light, the path is terminated and only the emitted radiance is added to the path. It can be thought of as lights having their BRDFs equal to zero.

[1] This particular value was taken from https://computergraphics.stackexchange.com/questions/5152/progressive-path-tracing-with-explicit-light-sampling .

The output of this algorithm is shown in figure and although it is correct, it can be improved which is the goal of chapter 6.

Algorithm 1 Path tracing Algorithm

$n \leftarrow 0$

$throughtput \leftarrow 1.0$ ▷ How much light can pass along the path
$LI \leftarrow 0$ ▷ Radiance arriving at the camera's pixel
$ray \leftarrow$ `GenerateCameraRay()`

while $n++ <$ MAX_PATH_LENGTH **do**
  $intersection \leftarrow$ `IntersectScene`(ray)
  if $intersection == None$ **then return** LI
  else if $intersection == Light$ **then**
    ▷ Transfer the radiance via the path to the camera.
    return $LI + throughput * intersection.LI$
  else ▷ Hit an object
    $\omega_o \leftarrow -ray.dir$ ▷ Points away from the surf.
    $\omega_i \leftarrow$`genNewDirection`$()$
    $costTheta \leftarrow \text{dot}(intersection.n, \omega_i)$
    $througput* = 2\pi * $ `BRDFEval`$(intersection, \omega_o, \omega_i) * costTheta$
  end if
  $q \leftarrow \max(throughput.x, throughput.y, throughput.z)$
  if $q <$ `random()` **then**
    $throughput/ = q$
  else
    break
  end if
end while

LI is not strictly necessary here because it will get set only once when a light is hit. But this formulation is closes to the theory and the improved algorithm shown later.

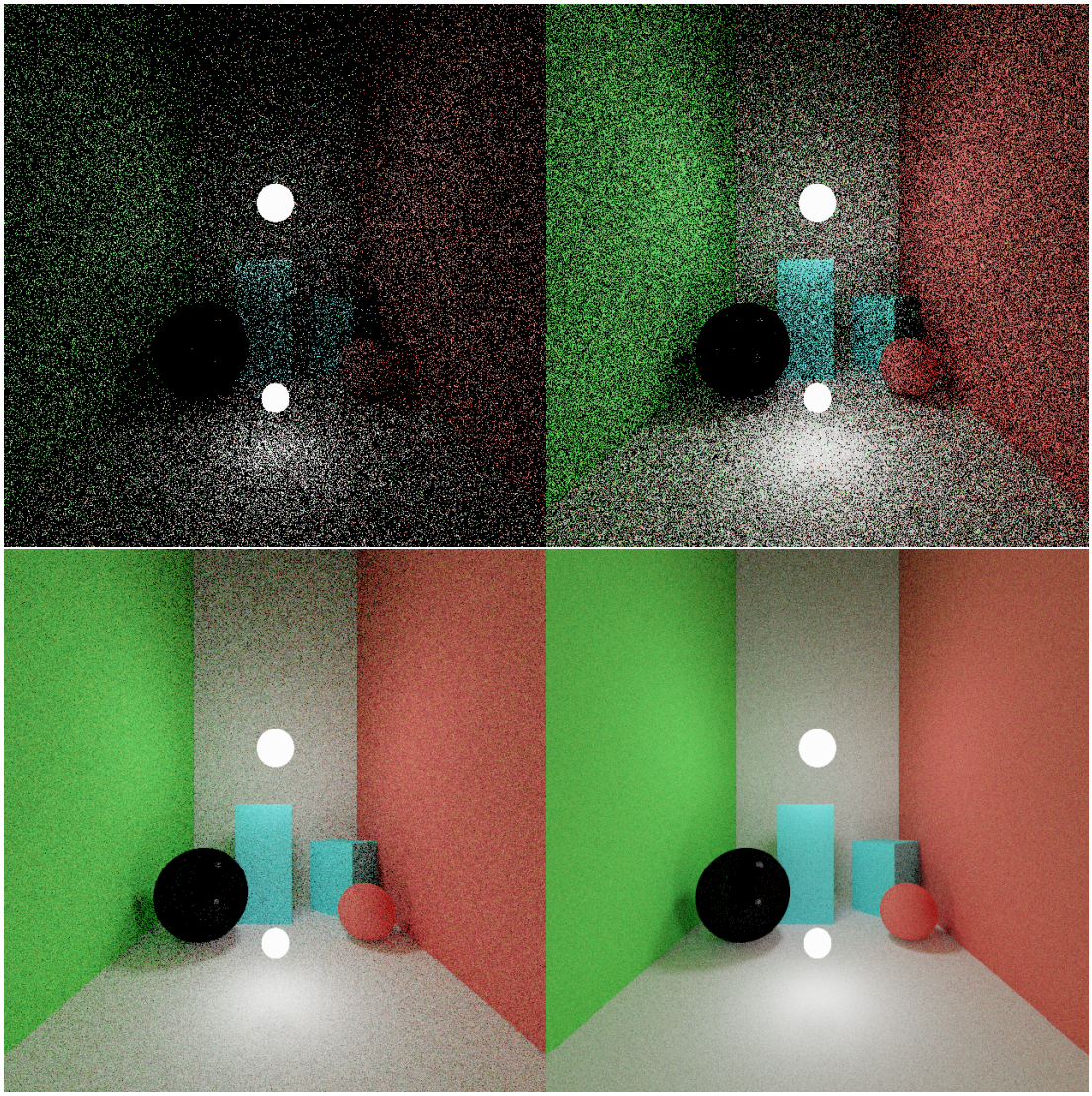

Figure 2.1: This figure show a Cornell-box-like scene rendered using the path tracing algorithm presented in section 2.4 with maximum numbers of 5 bounces. The images show (*from left to right, top to bottom*) 10, 50, 500, 2000 samples per pixel. Since only the path which hit the light are useful, it takes many samples to obtain a decent image. The left sphere uses Cook-Torrance material with $m = 0.01$ which makes it very mirror-like and the algorithm barely shows any reflection even from the lights.

3. Scene Representation

In the previous chapter 2 we derived first path tracing algorithm. This chapter will build the tools needed to determine if a light ray intersects any objects in the scene, and how can we generate rays originating from a camera. We can start by defining a ray which can be done by specifying its origin o and a direction d, every point on this ray is then given by eq. (3.1) and a parameter $t \in \Re_0^+$.

$$x = o + t.d; \tag{3.1}$$

3.1 Camera

Pinpoint camera was briefly introduced in section 1.4 as the object that represents the final result of the rendering algorithm. Reflecting the image plane in the figure fig. 1.3 creates a virtual receptive field. Generating rays emitted from the camera can then be done by choosing a pixel in the field through which send the ray; more precisely through a random point in this pixel . The camera can be represented by a world position p, three orthogonal vectors v, r, u for orientation and width, heigh - dimensions of the receptive field.

3.2 Objects

3.2.1 Sphere

A sphere object is parametrized by its centre c and a radius r, its surface is given by eq. (3.2). In order to determine if a ray intersect the sphere the eq. (3.1) is substituted into 17. Then solving the quadratic equation in term of t in eq. (3.3). Based on the number of positive roots the ray either did not hit the sphere or there's at least one hit point in which case we take the smaller one.

$$(x - c)^2 = r^2 \tag{3.2}$$

$$(o + td - c)^2 = r^2 \qquad O := o - c \tag{3.3}$$

$$(td - O)^2 = r^2$$

$$t^2 d.d + 2td.O + O.O - r^2 = 0$$

$$t_{1,2} = \frac{-d.O \pm \sqrt{(d.O)^2 - 4t^2(O.O - r^2)}}{2t^2}$$

$$n = O + t.d \tag{3.4}$$

$$t = min(t_1, t_2) \tag{3.5}$$

3.2.2 Plane

A plane can be defined by its normal vector n and any point p laying on that plane. The intersection point 't' can be found as

$$t = \frac{(p - o).n}{|n.d|}. \tag{3.6}$$

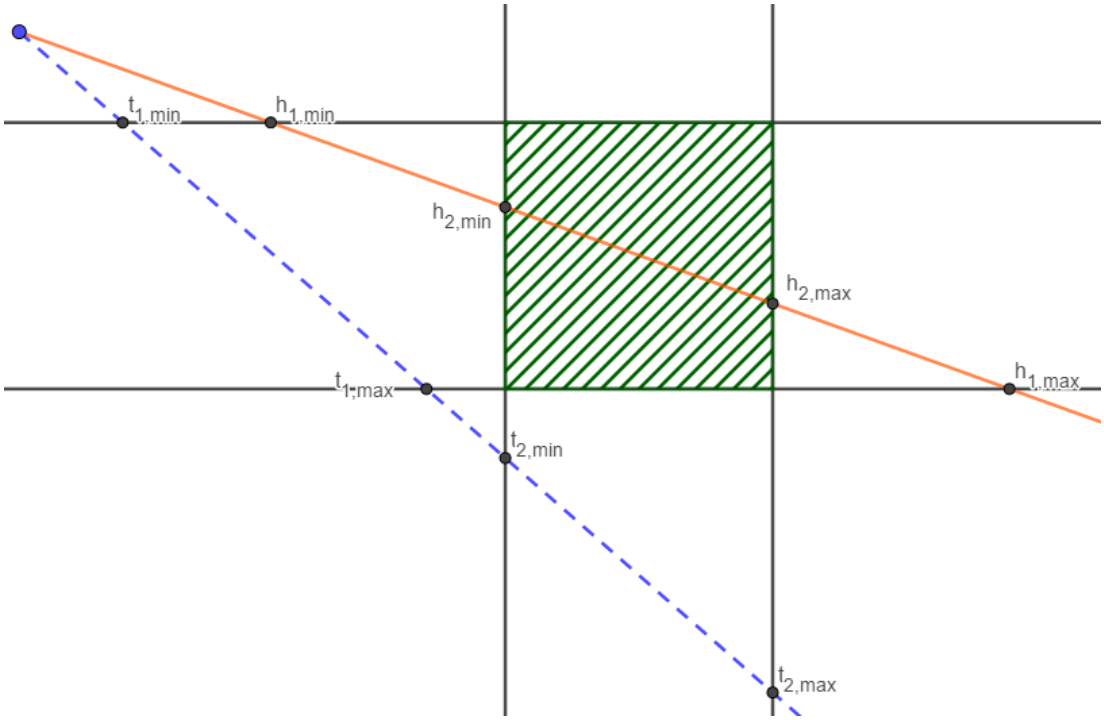

Figure 3.1: The *slab method* shown on two different rays. The solid one intersects the object and all it's minimal h values are less than any maximal value. On the other hand the dashed ray does not intersect the rectangle and indeed there exist $x = 1, y = 2$ such that $t_{1,max} < t_{2,min}$. Note that this method requires computing even negative t values in a case where the origin of the ray is between two planes.

Note that for rays that are parallel to the plane, the term $n.d$ is zero. Also $t < 0$ means that the plane is behind the ray. This method makes the plane two-sided is what we use in our program.

3.2.3 Cuboid

Cuboid can be defined by one point and three orthogonal vectors. The intersection test can be implemented by the *slab method* as presented in [7].

We will explain it in 2D with a rectangle, generalization to 3D is trivial. The method sequentially intersects the ray against the two pairs of parallel planes going through the sides of the rectangle. The algorithm is drawn in section 3.2.3.

For each pair $x \in \{1, 2\}$ we obtain two values of t - $t_{x,min}, t_{x,max}$ corresponding to the nearer and farther intersection, respectively. Then, following values are computed $t_{min} = \max(t_{1,min}, t_{2,min})$, $t_{max} = \min(t_{1,max}, t_{2,max})$ and the ray intersect the rectangle(cuboid) at point t_{min} if and only if $t_{min} \leq t_{max}$.. In other words, $t_{x,min} \leq t_{y,min}$ $x, y \in \{1, 2\}$. Care must be taken when the ray happens to be a parallel to any plane, in this case the values are set to infinities: $t_{x,min} = -\inf, t_{x,max} = \inf$. The source explains the method for axis-aligned rectangles but the idea holds for arbitrarily oriented ones provided the correct plane intersection method is used and also return points with t less than zero.

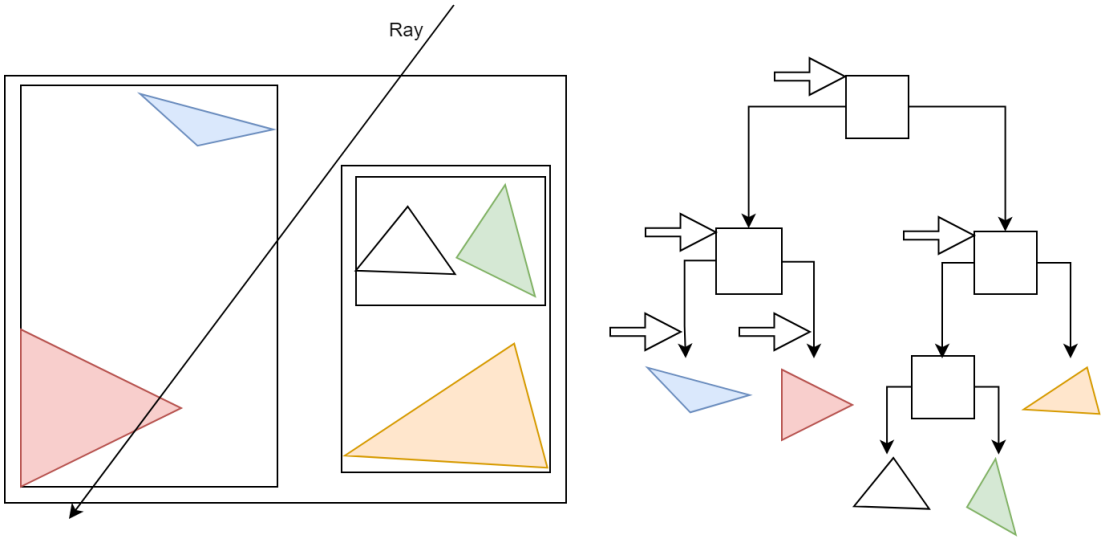

Figure 3.2: The bounding volume hiearchy shown on 5 triangles. The right image shows the tree structure, where internal nodes represent used AABBs as bounding boxes. Intersecting the ray with the scene consists of traversing the tree, intersection tests are marked with an arrow. In this case there are 5 of them which is equal to the testing the triangles individually.

3.2.4 Triangle and Triangle Mesh

A triangle can be characterized in more than one way e.g. using its three vertices or an vertex and two edges. For intersecting a triangle with a ray Moller-Trumbore algorithm[8] is used. It has simple implementation and offers reasonable speed which is needed for the next object type.

Triangle meshes are widely used to model 3D objects in computer graphics. Over the years GPUs became highly optimized in processing and rasterizing these triangles. Meaning models having a million or even more of them are not uncommon.

Naive implementation of a ray and the triangle mesh intersection would test the ray against all the triangles and reported the closest intersection. In our application this approach was not really feasible for more than a hundred triangles. A great deal of research went into the development of accelerating structures that offer better than $O(n)$ intersection test. Their implementation on CPUs and recently also on GPUs e.g. [9].

One such structure is a Bounding volume hiearchy(BVH) which is a hierarchical structure that groups close objects (in our case triangles) into a bounding volume(BV), these volumes are then clustered again, thus building a tree of them. The general idea is shown in section 3.2.4. A balanced tree containing n triangles can be intersected with a ray in $O(logn + \text{number of intersections})$ time which is typically much faster than a naive $O(n)$ solution. Intersecting a ray with the BVH starts by testing the root node's BV and in case of a hit recursively testing its children. This way if the BV of an internal node does not intersect the ray whole sub-tree is not visited.

Our application uses a BVH described in [10]. The paper uses axis-aligned

bounding boxes(AABBs) as the bounding volumes and only considers binary internal nodes; leaf nodes may contain more than one triangle. They are build from top to bottom using a *surface area heuristic*(SAH) and centroid-based partitioning.

This algorithm starts with an array of n individual triangles wrapped inside their AABBs and one AABB that envelopes the whole array. These bounding volumes are sorted according to their centers from left to right for all three axes. Then a binary split is considered at each possible index splitting the array into two groups, the SAH is used to rank these splits and the split with lowest SAH is chosen. A node is created with the one AABB and two children. These children are created by splitting the two subarrays and enveloping them in another two AABBs. This recursive top-to-bottom algorithm either stops when the array only contains one element or when the SAH cost of any split is greater than the cost of creating a leaf node for the elements.

The paper uses the following formulas for calculating the cost of a split and of a leaf node:

$$T_{split} = 2T_{AABB} + \frac{Area(S_1)}{Area(S)} |S_1| * T_{tri} + \frac{Area(S_2)}{Area(S)} |S_2| * T_{tri},$$

$$T_{leaf} = |S| * T_{tri}.$$

$S, S_1, S_2 \ldots$ The splitted array and the two resulting groups of triangles.

$Area(S) \ldots$ Surface area of the AABB enveloping the set S.

$T_{AABB}, T_{tri} \ldots$ Cost of intersection test with a AABB and a triangle.

Traversing this tree means doing an intersection with the node's children' AABBs and recursing into the intersected ones. We can reuse the cuboid intersection for the AABB which can be further optimized given the normals are aligned to axes as was originally presented in[7].

3.3 Lights

Our scene supports 4 types of lights. Point lights have infinitesimal size and produce hard shadows, their implementation requires *next event estimation* technique that will be described lated. Rectangular lights are capable of producing soft shadow which greatly increases the realism in scenes. Similiar to these are sphere lights which might be better to suited in some scenes as they shine in all directions. The last type of light is an environment light in the form of HDR map. This map represents light coming from distant objects which are not part of the scene. This map can act as a simple background light or it can be the sole source of the light in the scene.

4. Overview of BRDF models

In this chapter we introduce a few well-known models of reflectance functions. More exhausting overview can be found in [11],[5].

Note that all calculations are done in world-coordinates and are valid only for reflected rays i.e $(\omega_i.n), ()\omega_o.n) \geq 0$.

In general, the reflectance function is dependent on wavelength λ of the light. This dependency can be approximated by the RGB model which evaluates it for 3 wavelengths corresponding to red, green and blue light. This correspond to human eye's sensitivity, for more thorough explanation see e.g. [5]. The reflectance functions introduced here deal with it through defining $k \in [0,1]^3$ factors which characterize the portion of light scattered and not absorbed. They can be refered to as the objects colours or albedos.

4.1 Lambert

Lambert BRDF represents a perfectly diffuse material that reflects a portion of incoming light uniformly in all directions. The BRDF is named after Johann H. Lambert who described this material long before the field of computer graphics.[1]

Given $f(x, \omega_i \rightarrow \omega_o) = c$ for some $c \in \mathbb{R}$. The conservation of energy law dictates that

$$\int f(x, \omega_i \rightarrow \omega_o)(n.\omega_i)d\omega_i = 1 \tag{4.1}$$

$$\int c(n.\omega_i)d\omega_i = 1 \tag{4.2}$$

$$c\pi = 1 \quad \rightarrow \quad c = \frac{1}{\pi} \tag{4.3}$$

$$f(x, \omega_i \rightarrow \omega_o) = \frac{1}{\pi} \tag{4.4}$$

Factor $k_{diffuse}$ can be added for coloured objects. In that case $f(x, \omega_i \rightarrow \omega_o) = \frac{k_{diffuse}}{\pi}$.

4.2 Functions based on microfacet theory

4.2.1 Microfacet theory

The concept introduced by Torrance and Sparrow in [12] states that rough materials can be thought of as consisting of many differently-oriented small mirrors - *microfacets* - which obey the law of reflection.

Incoming light then can either be absorbed, scattered which is described with Lambert model - *diffuse term* - or it can be reflected by these mirrors - *specular term*. Thus the BRDF can be computed as $f_r = f_d + f_s$ and $f_r = k_{diffuse}f_d + k_{spec}f_s$ is often used for explicit colors and also to ensure conservation of energy. [13] proposed to model their orientation with probability distribution $D(\alpha)$ describing

[1] `https://en.wikipedia.org/wiki/Lambertian_reflectance`

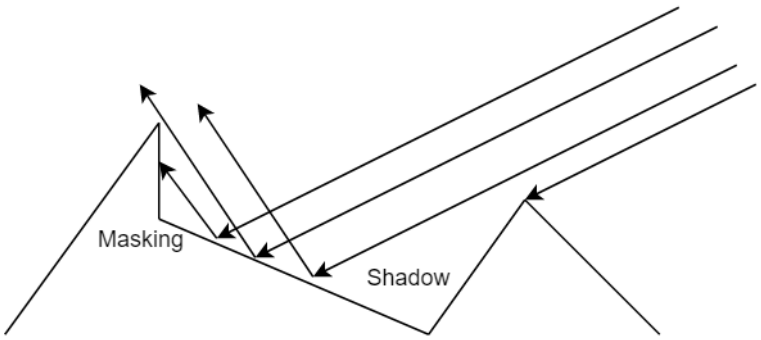

Figure 4.1: Shadowing and masking in the microfacet model. The incoming light might not hit the full microfacet and some light won't hit the viewer because they too cannot see the full facet. This behaviour is approximated by the G term.

deviation of mirror's normal from the direction $h = (\omega_i + \omega_o)$ known as halfway vector. Due to law of reflection a mirror's normal must be equal to h in order to reflect light from ω_i to ω_o. The paper then describes a geometrical factor G that accounts for photons being either occluded before hitting a mirror or after and thus not leaving the surface in the ω_o direction. See fig. 4.1 for the idea. This geometrical factor depends on the orientation of the microfacets and thus on D, [13] give a thorough explanation for their choice of G.

4.2.2 Cook-Torrance

[13] uses the microfacet theory to define the Cook-Torrance model with the specular term f_s shown in eq. (4.5) together with G and D. The paper explicitly mentions using two distributions Beckmann or Blinn's Gaussian distribution for D. Where m is a parameter called roughness and $\alpha = \cos^{-1}(n.h)$.

Note that the original paper seems to lack the $\frac{1}{4}$ term in f_s, which was also cited in [14], [11]. But according to derivation done in [5] it should be included.

$$f_s = \frac{FDG}{4(n.\omega_i)(n.\omega_o)} \tag{4.5}$$

$$G = \min(1, \frac{2(n.h)(n,\omega_o)}{h.\omega_o}, \frac{2(n.h)(n.\omega_i)}{h.\omega_o}) \tag{4.6}$$

$$D_{Beckmann} = \frac{1}{m^2 \cos^4 \alpha} e^{-(\frac{\tan \alpha}{m})^2} \tag{4.7}$$

$$D_{Gaussian} = ce^{-(\frac{\alpha}{m})^2} \tag{4.8}$$

F stands for Fresnel term. It is the solution to Fresnel equations which describe the relationship between refracted and reflected light on an object's surface. In general, they depend on polarization of incoming light, indices of refractions, and type of the material. We give the approximation presented in the paper [13]:

$$F = \frac{1}{2}(\frac{g-c}{g+c})^2(1 + (\frac{c(g+c)-1}{c(g-c)+1})^2)$$
$$c = \omega_i.h$$
$$g = n^2 + c^2 - 1.$$

Full equation can be found in [15]. Another option is to use Schlick's approximation from the same paper:

$$F = F_0 + (1 - F_0)(1 - \omega_i.n)^5$$
$$F_0 = (\frac{n_\lambda - 1}{n_\lambda + 1})^2,$$

where n_λ is index of refraction(i.o.r.) of the material. The 1 corresponds to i.o.r. of medium, in this case air. We explicitly highlighted that i.o.r dependence on wavelength of the incoming light. In the RGB model n_λ would be a 3-dimensional component vector.

4.2.3 Ashikhmin-Shirley

Ashikhmin-Shirley [16] is another microfacet model but this one can exhibit anisotropic properties. Meaning that the reflected light is not only dependent on elevation angles of θ_i, θ_o but also on azimuth angles Φ_i, Φ_o. Relevant equations for the model given from the paper are summarized in eq. (4.9). Parameters n_u, n_v controls the anisotropicity of the material, u, v are tangent and bi-tangent vectors. Notice that for $n_u = n_v$ the numerator in β reduces to n_u. The paper used Lambert model for the diffuse part and added correction factor to ensure energy conversation. They also recommend using the Schlick approximation for the Fresnel term. h is again the half vector.

$$f_s = \frac{\sqrt{(n_u + 1)(n_v + 1)}}{8\pi} F(\omega_i.h) \frac{(n.h)^\beta}{(\omega_i.h) max(n.\omega_i, n.\omega_o)} \tag{4.9}$$
$$\beta = \frac{n_u(h.u)^2 + n_v(h.v)^2}{1 - (n.h)^2}$$
$$f_d = \frac{28k_d}{23\pi}(1 - k_s)(1 - (1 - \frac{n.\omega_o}{2})^5)(1 - (1 - \frac{n.\omega_i}{2})^5)$$

4.2.4 Oren-Nayar

Oren-Nayer model created by [17] also uses the microfacet theory but instead of facets being mirrors, they are lambertian surfaces. The paper describes isotropic and anisotropic distributions of these facets. The final model is given as

$$f = \frac{k_{diffuse}}{\pi} \cos\theta_i (A + B. \max(0, \cos(\phi_i - \phi_o)). \sin\alpha \tan\beta),$$
$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33},$$
$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09},$$
$$\alpha = \max(\theta_i, \theta_o),$$
$$\beta = \min(\theta_i, \theta_o).$$

Where $\sigma \in \mathbb{R}^+$ corresponds roughness of the surface. This formulation contains more than one goniometric function which is not ideal. Jos van Ouwerkerk on

his blog [18] modified the formula to use only vector math. The end result is

$$K = \max(0, n.\omega_i) \qquad V = \max(0, n.\omega_o),$$
$$P = \max(0, (||\omega_i - K.n||.||\omega_o - V.n||)),$$
$$f = \frac{k_{diffuse}}{\pi}(A + B.P\frac{\sqrt{(1-K^2)(1-V^2)}}{max(K,V)}.$$

It only requires dot products, square roots and two normalizations.

4.3 Measured models

The previous sections dealt with analytical models for the reflectance functions but their goal is to resemble real-world materials. Another approach taken by [19] is to measure real-world objects under precise lighting conditions to obtain tabular values for the BRDF. Their measurements have granuality of 1° for elevations angles and ϕ_o. That means each BRDF has almost 1,5 million samples. They noted that only $\phi_o \in [0, 180^\circ]$ is necessary due to the reciprocity rule.

Their approach can successfully model many different materials but we do not explore this area further and the application does not support measured BRDFs. The reason is that there is a little value in "editing" these functions.

The second part of [19] uses the measured models to create a 15-dimensional space of reflectance functions. That, on the other hand, would be an interesting topic and might be worth-while exploring, but currently it is only in the section for future work.

5. Random Numbers and Monte Carlo Integration

In this chapter we will revisit Monte Carlo integration and take a look at generating random numbers on computers. Previously, it was stated that the definite integral in eq. (2.2) of the function $f(x)$ can be estimated as the average of $f(\ddot{x}_i)$ values for uniformly-distributed random numbers $\ddot{x}_i$. This approach can, in fact, be generalized to any probability distribution $p(x)$ as long as the following condition holds

$$\forall x \in \Omega : |f(x)| > 0 \implies p(x) > 0.$$

This generalization is very useful because it can greatly reduce estimator's variance for the given number of samples. The estimate is then computed as eq. (5.1) and the previous uniform case corresponds to $p(x) = \frac{1}{C}$

$$\frac{1}{N}\sum_{i=1}^{N}\frac{f(\ddot{x}_i)}{p(\ddot{x}_i)}. \tag{5.1}$$

Ideally we would like to choose p proportional to f, reasoning behind that can be seen by choosing exactly the $p(x) = \frac{f(x)}{A}$. Doing so, leads to the estimate for the integral in eq. (2.2) as

$$\frac{1}{N}\sum_{i=1}^{N}\frac{f(\ddot{x}_i)}{\frac{f(\ddot{x})}{A}} = \frac{1}{N}\sum_{i=1}^{N}A = A$$

That means we get the exact result for any numbers of samples. Of course, this is only an illustration, if A is known in the first place, there is no need to estimate it. But picking $p \propto f$ is to some degree possible and will lead to reduced variance in the estimate. Note that the opposite is also true, choosing p different from $f(x)$ can lead to worse estimates than using uniform distribution. They will still be unbiased but it can take very large number of samples to get close to the true value.

5.1 Random Numbers on Computers

Modern processors are capable of generating truly random numbers using special hardware instructions e.g. `RDRAND` [20][21], but as far as we are aware, there are no graphics cards capable of this or any other similar instructions. Instead, considerable time went into researching and effectively implementing pseudo-random generators that can generate numbers that appear random under many statistical tests. The common well-known generators include *Mersenne Twister* and families of *linear congruential generators*. Another branch is to use low-discrepancy sequences that generate quasi-random numbers, quick overview of them can be found in [6] and it was also the approach taken by [22].

For our purposes we will consider two generators - *MWC64X* [23], and *PCG-XSH-RR* from the *permuted congruential generator*(PCG) family[24]. Both can

be very easily implemented inside a GPU and require very little state in contrast to the Mersenne Twister. Authors claim, in the latter case explain [24], that both are of high-quality and should pass many statistical tests for randomness. Periods of these algorithms are 2^{63} and 2^{62}, respectively, more than enough for our needs. During our testing we did not see any difference between them.

Algorithm 2 Pseudo-random number generators

Input $S \ldots$ 64-bit state
Output Uniformly distributed floating-point number in$[0, 1]$

procedure MWC64X
    $c \leftarrow S >> 32$
    $x \leftarrow S\&xFFFFFFFF$
    $S \leftarrow 6364136223846793005x + c$
return $x \oplus c$
end procedure

procedure PCG
    $x \leftarrow S >> 59$
    $r \leftarrow (S \oplus (S >> 18)) >> 27$
    $r \leftarrow (r >> x)|(r << (32 - X))$
    $S \leftarrow 6364136223846793005S + 1442695040888963407$ **return** r
end procedure
Lower-case letters are 32-bit numbers, upper-case 64-bit. The constants are taken from [25]. Other multiplicative constants are possible according to the section 4.3.3 of [24] and the additive factor can reportedly be an arbitrary odd number.

6. Improvements to the Path Tracing algortihm

The first version estimates LTE correctly but as shown in fig. 2.1 doing so require many samples. This chapter will present existing techniques to reduce the variance in the estimation. One such technique was presented in chapter 5, others include *Next event estimation* and *Multiple importance sampling*.

6.1 Next Event Estimation

The approach that will be described was presented in [26] and given a name Next event estimation in [4]. But similar idea was very briefly mentioned in [3] and the direct lighting techniques is also discussed in [2]. The main reason why the algorithm presented in section 2.4 does so poorly in indoor scenes is that the returned L is non-zero only if we happen to hit a light, otherwise the computation was wasted. Instead, [4] takes the area form of the LTE from eq. (1.10) and replaces the second term with L_r standing for the *reflected light*. By substituting the integral into itself the paper arrives at the form written in eq. (6.1). The Last equality is the consequence of linearity of the integral hidden inside R.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_i) \tag{6.1}$$
$$L_o(x, \omega_o) = L_e(x, \omega_o) + R[L_{i,e} + L_{i,r}](x, \omega_i) \tag{6.2}$$
$$L_o(x, \omega_o) = L_e(x, \omega_o) + R[L_{i,e}](x, \omega_i) + R[L_{i,r}](x, \omega_i) \tag{6.3}$$

Notation $L_{i,e}(x, \omega_i)$ represents radiance received from direction $-\omega_i$[1] that was emitted from the closest point from x in the direction ω_i. I.e the point y in the area integral form and so $L_{i,e}(x, \omega_i) = L_e(y, -\omega_i)$. Analogously for reflected radiance $L_{i,r}(x, \omega_i)$.

This form allows us to estimate the two integrals independently of each other and we can choose the preferred form for each of them. The middle term is non-recursive integral corresponding to direct illumination, $R[L_r](x, \omega_i)$ is still recursive and represents the incoming radiance at the point x that was not directly emitted from the closest hit-point but rather reflected from yet another source.

$R[L_{i,r}]$ is estimated analogously as in eq. (2.3a), using the solid angle which is more useful and natural form than using area form and choosing a random point that might not even be visible.

Estimating $R[L_e]$ using solid angle formulation would not be very useful as only those rays that hit a light would be non-zero. That should hint towards using the area formulation. First, the integral can be divided into $|\mathbb{A}|$ integrals

[1] Still using the convention $L_o(x, \omega) = L_i(x, -\omega)$ with ω point away from x.

over individual areas

$$R[L_{i,r}] = (x, \omega_i) \int\limits_{A_y \in \mathbb{A}} L_e(y, -\omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) \frac{-n_A.\omega_i}{||y-x||^2} V(x,y) dA_y =$$
$$= \sum_{i=1}^{|\mathbb{A}|} \int\limits_{A_y} L_e(y, -\omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) \frac{-n_A.\omega_i}{||y-x||^2} V(x,y) dA_y =$$
$$= \sum_{i=1}^{|Lights(\mathbb{A})|} \int\limits_{A_y} L_e(y, -\omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) \frac{-n_A.\omega_i}{||y-x||^2} V(x,y) dA_y$$

and the last equality holds because only the integrals over lights' surfaces can be non-zero. The estimation of $R[L_{i,r}]$ is summarized in eq. (6.4) (optionally a different N can be used for each light).

$$R[L_{i,r}] = \sum_{i=1}^{|Lights(\mathbb{A})|} \int\limits_{A_y} \frac{Area(A_y)}{N} \sum_{k=1}^{N} L_e(y, -\omega_i) f(x, \omega_i \to \omega_o)(\omega_i.n) \frac{-n_A.\omega_i}{||y-x||^2} V(x,y) dA_y \tag{6.4}$$

This reformulation of the LTE from [4] can drastically improve the convergence speed of the path tracing algorithm which is used in section 6.4. We again only use $N = 1$ for estimating the direct lighting. Notice that we no longer add the radiance if we hit a light, doing so would cause that term to be added twice as it is already included in the previous iteration. The exception is hitting a light directly with the camera ray, in that case we have to add it. Not doing so would make the lights invisible to the camera.

Note than a welcomed consequence of this strategy is the ability to use point lights. The basic version of path tracing will never hit a point light and so its energy can never be transferred to the image plane.

This addition greatly improves the convergence speed as can be seen in fig. 6.1 compared to fig. 2.1.

6.2 Applying importance sampling to LTE

Previously inchapter 5, it was stated that by using suitable probability distribution to generate random numbers we can reduce the variance of integral estimates. This section will apply this technique to various parts of the path tracing algorithm in order to achieve faster convergence speed.

6.2.1 Reflected Radiance

Estimate of the radiance from the reflected light is given as

$$L_o(x, \omega_o) = \frac{1}{N} \sum_{i=0}^{N} \frac{L_{i,r}(x, \ddot{\omega}_i) f(x, \ddot{\omega}_i \to \omega_o)(\ddot{\omega}_i.n)}{p(\ddot{\omega}_i)}$$

.

Finding $p(w)$ that is proportional to all the included terms is hard. Mainly because we simply do not know anything about $L_{i,r}$ dependence on ω_i for the given

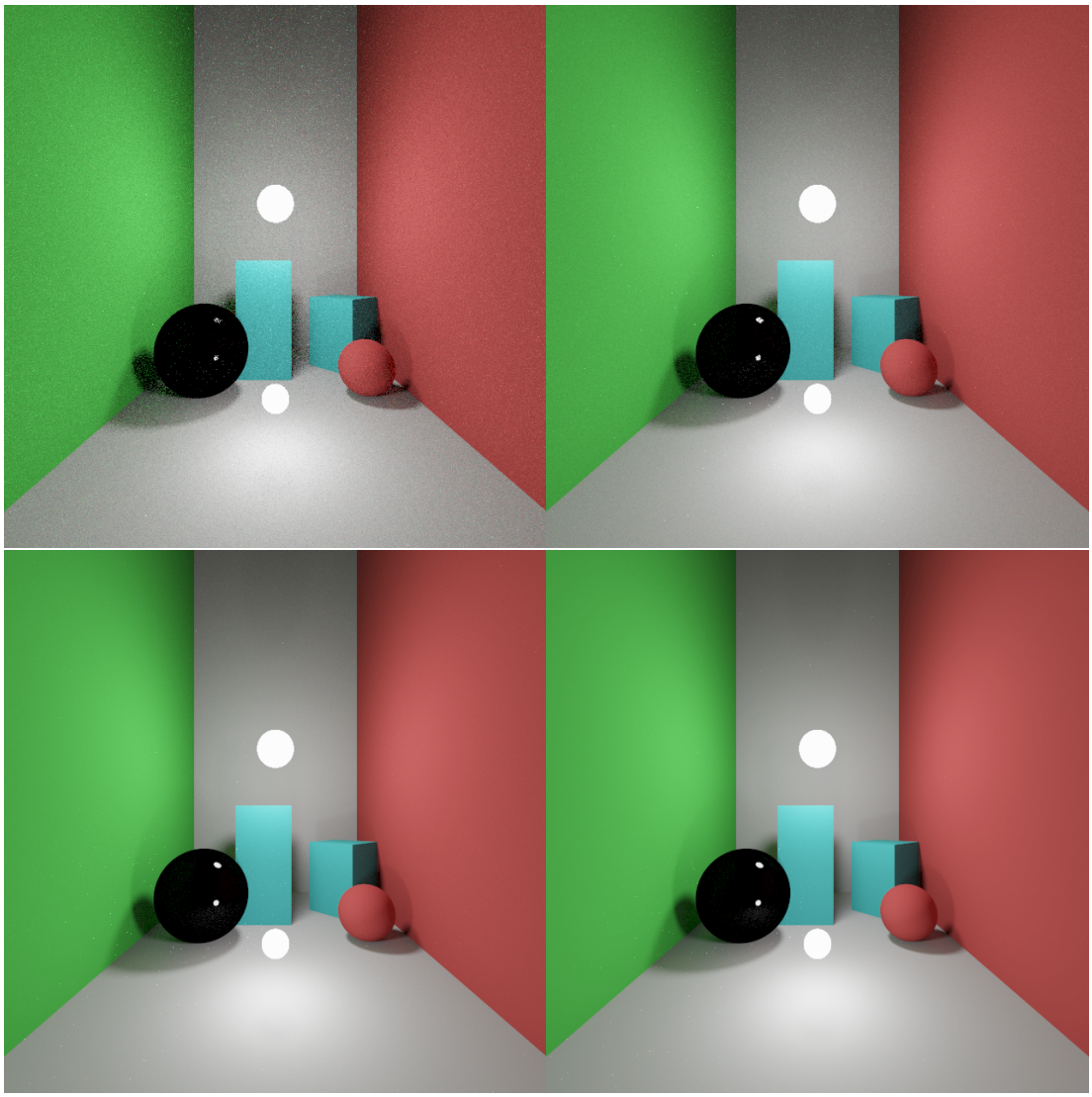

Figure 6.1: This figure show a Cornell-box-like scene rendered using the next event estimation. The images show (*from left to right, top to bottom*) 10, 50, 500, 2000 samples per pixel. In contract to images in fig. 2.1, this added method converges much more quickly at least for small lights. But it still has problems with showing reflections in the sphere. It only shows the lights as they were sampled directly, but not the floor and walls.

x, at least not in a computable way. The easiest solution is to also disregard the BRDF term as they too can be complicated. What is left is $(\ddot{\omega}_i.n)$ and sampling according to this distribution is called cosine-weighted sampling.

6.2.2 Cosine-weighted sampling

Instead of uniformly picking a direction in $\mathbb{H}^2$ with $p(\omega) = \frac{1}{2\pi}$ we can choose ω with $p(\omega) = \frac{\cos(\omega)}{\pi}$. The π term is normalization factor to make p a proper probability distribution. Sampling from p is in fact very easy, given two uniform random numbers $u_1, u_2 \in [0, 1]$, surface normal n, and two orthogonal vectors t_1, t_2, the resulting $\omega = (x, y, y)$ is given by

$$x = \sqrt{u_2}\cos(2\pi u_1)t_1 \qquad y = \sqrt{u_2}\sin(2\pi u_1)t_2 \qquad z = \sqrt{1-u_2}n$$

According to [5] this sampling formula is know as *Malley's method*. The book then says: *The idea behind Malley's method is that if we choose points uniformly from the unit disk and then generate directions by projecting the points on the disk up to the hemisphere above it, the resulting distribution of directions will have a cosine distribution.* We use this as the default sampling strategy for unknown BRDFs in estimation of $R[L_{i,r}]$.

6.2.3 Sampling according to BRDF

It can be seen that the reflectance functions presented chapter 4 might not be very uniform and rather might have very sharp peaks and so a uniform distribution might not be very appropriate. The reflectance function itself is almost a probability distribution if it obeys the established laws, so ideally we would choose $p = c.f_r$; c being the normalization factor. But sampling from these distributions is not straightforward, for that reason authors often provide [11] sampling strategies that are at least similar to their functions. We will list them for the reflectance functions introduced in chapter 4.

Mirror Custom sampling allows us to define a BRDF for perfect mirrors - materials that obeys the law of reflection. Although we already encountered mirror-like surface in the microfacet models, for them the $L_i(x, \omega_i)$ term was non-zero for non-trivial number of incoming directions. In this case there's only one such direction - $\omega_i = -d.n + 2(d.n)n$. Randomly choosing this vector has practically zero probability 0f happening. Using the importance sampling we can directly choose this vector with probability 1 and the integral has been reduced to one evaluation.

Lambert Is a uniform distribution without any preference for a particular ω and so cosine-weighted sampling is better for evaluating the LTE.

Cook-Torrance The dominant term in the specular part is the D distribution of microfacets. So the reasonable approximation is to sample according to it and this will be now presented as was described in [5].

The approach is to first generate the half-way vector ω_h and then transform it to ω_i. Let $u_1, u_2 \in [0, 1]$ be two uniform random numbers. Since this BRDF is

isotropic the azimuth angle ϕ_h can be sampled uniformly as $\phi_h = 2\pi u_1$. Next it is easier to obtain ω_h indirectly as

$$\tan^2 \omega_h = -m^2 \log(u_2) \tag{6.5}$$

$$\cos \omega_h = \sqrt{\frac{1}{1 + \tan^2 \omega_h}} \tag{6.6}$$

$$\sin \omega_h = \sqrt{1 - \frac{1}{1 + \tan^2 \omega_h}} \tag{6.7}$$

$$. \tag{6.8}$$

Using these we can construct the vector ω_h. For $p(\omega_h)$ the D can be used directly. The last thing that remains is to get ω_i and that can be done as

$$\omega_i = \omega_o + 2(\omega_h, \omega_o)\omega_h \tag{6.9}$$

$$p(\omega_i) = \frac{\omega_o}{4 * \omega_o.\omega_h}. \tag{6.10}$$

Ashikhmin-Shirley [16] proposes following the sampling strategy, given two random numbers $u_1, u_2 \in [0, 1]$ generate the halfway vector $h = (\theta_h, \phi_h)$ as

$$\phi_h = tan^{-1}(\sqrt{\frac{n_u + 1}{n_v + 1}} tan(\frac{\pi}{2} u_1))$$

$$\cos \theta_h = (1 - u_2)^{-n_u \cos^2 \phi_h - n_v \sin^2 \phi_h}.$$

$$p(h) = \frac{\sqrt{(n_u + 1)(n_v + 1)}}{2\pi} (n.h)^{n_u \cos^2 \phi_h + n_v \sin^2 \phi_h}$$

The source then notes that ϕ_h is generated only in $0, \frac{\pi}{2}$ and must be mapped to $0, 2\pi$, see details in the paper. Lastly, this generates the halfway vector, ω_i can be obtained[16] as $-\omega_o + 2(\omega_i.h)h$ and $p(\omega_o) = \frac{p(h)}{4\omega_o.h}$.

Oren-Nayar is close enough to Lambertian surface so that it can too be sampled using the cosine-weighted distribution.

The impact of these improvements can be seen in figures fig. 6.2.

6.3 Direct Lighting

Although we cannot reason about the term $L_{i,r}$, we know more about $L_{i,e}$. Our scene consists of point, area, and sphere lights as well as one environment light.

Point lights have infinitely small area that only consists of one point. There is no advantage of using Monte Carlo as the corresponding integral in eq. (6.4) can be computed explicitly.

Points on area lights can be reasonably well sampled uniformly [figure with lambert]. There would be slight advantage to use $p \propto (\omega_i.n)\frac{-n_A.\omega_i}{||y-x||^2}$ but we won't do that yet.

Sphere lights are similar to area lights but a worthwhile optimization is to use cosine-weighted sampling strategy that samples the half of the sphere oriented

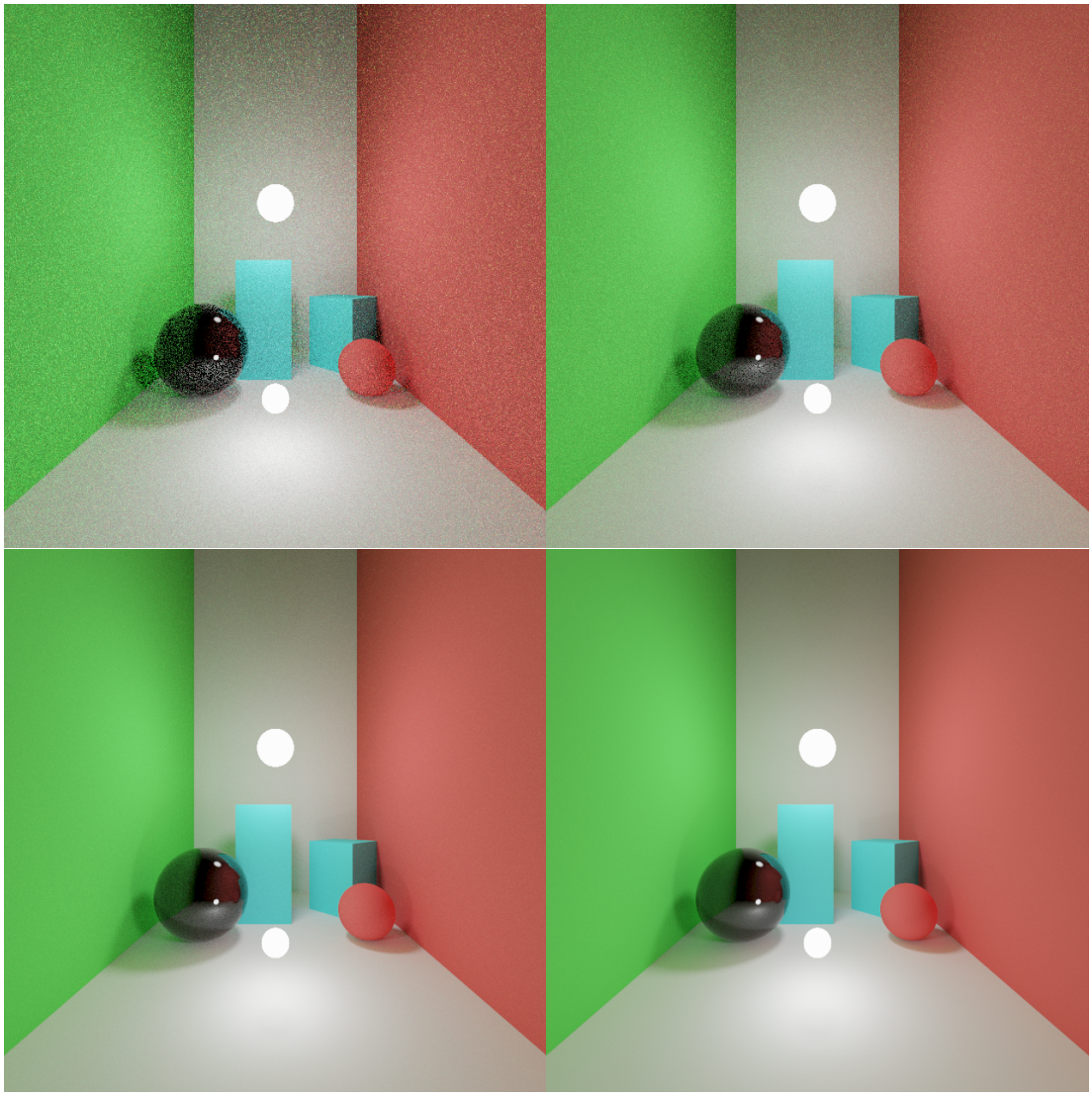

Figure 6.2: This figure show the same indoor scene now rendered with included BRDF sampling and MIS of light. The images show (*from left to right, top to bottom*) 10, 50, 500, 2000 samples per pixel. In contract to the images in fig. 6.1, sampling according to the BRDF can handle reflections of walls and light sampling preserves reflections of lights.

towards the hit-point x. Because $L_{e,i}$ is non-zero only on this half and its magnitude is given by $E_A(y)(-n_A.\omega_i)$ where $E_A(y)$ is irradiance emitted from the light's surface.

Sampling the environment light with special distribution instead of $p(\omega) = \frac{1}{4\pi}$ is very useful technique that further reduces the variance in the final image and will be more thoroughly described in following section.

6.3.1 Importance Sampling of Environment Light

Our environment light uses a 2D texture for calculating the incoming radiance to the scene from environment sources. That makes $L_{i,e}$ piece-wise constant[2] over the solid angle. Since it envelops the whole scene, there's a little advantage to using area form and instead the solid angle variant will be used.

Ideally, we want to choose a distribution as close to as $p(\omega) = \frac{L_{i,e}(y,\omega)}{\int L_{i,e}(y,\omega)d\omega}$ (optionally including the cosine term). Furthermore, due to the $L_{i,e}$ discretization, the integral can be computed directly by summing over the texture's pixels.

There are multiple methods how sample from a discrete distribution, overview of some is e.g. in section 13.3 of [5]. The basic method is the inversion transformation method. That is a computing cumulative distribution function(CDF) $F_u = P(\omega \leq u)$ (u is discrete variable), inverting it and then for given a $\ddot{\omega}$ sampled uniformly[3] $\acute{\omega} = F^{-1}(\ddot{\omega})$ has the p distribution. But sampling from the F^{-1} naively takes $O(n)$ time, binary search will reduce it to $O(\log n)$ for n discrete values. There is a paper [27] from 2008 that presents two new methods, first that runs in $O(1)$ time on average. Second that inverts the CDF approximately. We tried to implement it, but we have run into issues with indices as the paper does not specify behaviour for out of bounds indices which might happen. It is also very brief on the resulted algorithm, in particular choosing a row using rational numbers.

We have chosen to use *square histogram* method presented in [28]. Given a 1D discrete distribution defined using table 6.1 and visualized in a histogram fig. 6.3. The paper proposes to reorganize this histogram in such way that each column is exactly the same height - the average height of $\frac{1}{n}$ for n discrete values. They explain that this can easily be achieved by taking the highest column H and moving a part of it to the lowest column L such that L reaches the height $\frac{1}{n}$.. The highest column might become smaller than the average but since it's bigger than L it can always "*satures*" B to the desired height. Thus each column contains at most two different values.

The source gives a full algorithm for building this modified histogram and subsequently sampling according to the p distribution. We rewrote it using a slightly different notation in algorithm 3, the paper uses K, V tables to store the modified histogram. K_i stores the second value or i if there isn't one. V acts as a sort of cumulative distribution function. At first we tried this approach and did not manage to get it working (section 6.3.1). We decided to make a slight modification, instead of computing V_i we compute W_i which is defined as portion the column H occupies in a now-average-sized L. In other words, we have

[2]Depending on chosen interpolation between two texture samples it can also be bilinear or bicubic.

[3]With respect to the solid angle over the surface of a sphere

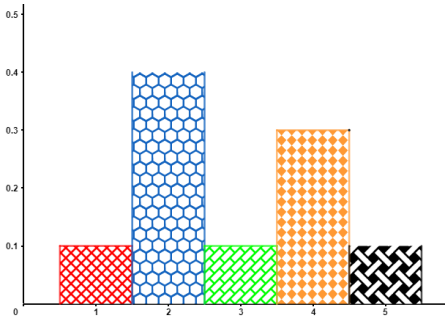

Figure 6.3: Original histogram

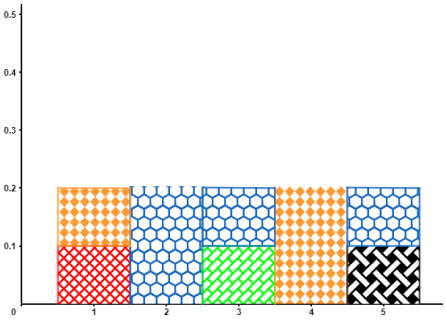

Figure 6.4: Square histogram

Figure 6.5: Histograms before and after the application of the square histogram method on the distribution defined in table 6.1.

split the 1D discrete probability into a uniform distribution over n values and n binary distributions over i, K_i defined by $p(K_i) = W_i$. Although this requires two random numbers rather than one, generating them using algorithms presented in section 5.1 is pretty fast and does not create a bottleneck in our application.

Later we managed to get a look at the supplementary material for the paper. The implementation uses a different formula $V_L = L.\alpha + p_L = (L+1).\alpha - s$. With this, the algorithm works as expected. The table 6.1 contains computed values by the presented algorithm on our example, fig. 6.4(right) shows the modified histogram. The table also shows the suspicious values of V_i and their fixed counterparts.

Images in section 6.3.1 show an environment map and 100 million samples generated using the square histogram method. To sample from 2D distribution the well-known approach is taken. That is we create a marginal distribution over rows and conditional distributions for each row over columns given the selected row. We then apply the algorithm to all of them. This approach is also used in our application. One last thing needed is that this probability is defined over $[0, w) * [0, h)$ (width and height) but the environment light sampling is done with respect to solid angle. Because of that the probability must be multiplied by the correction factor $\frac{w}{4\pi}$.

i	0	1	2	3	4
p_i	$\frac{1}{10}$	$\frac{4}{10}$	$\frac{1}{10}$	$\frac{3}{10}$	$\frac{1}{10}$
K_i	4	2	2	4	2
V_i	$\frac{-1}{10}$	$\frac{4}{10}$	$\frac{3}{10}$	$\frac{6}{10}$	$\frac{7}{10}$
$V_{i,fix}$	$\frac{1}{10}$	$\frac{4}{10}$	$\frac{5}{10}$	$\frac{6}{10}$	$\frac{9}{10}$
W_i	$\frac{1}{2}$	1	$\frac{1}{2}$	1	$\frac{1}{2}$

Table 6.1: 1D discrete distribution with CDF.

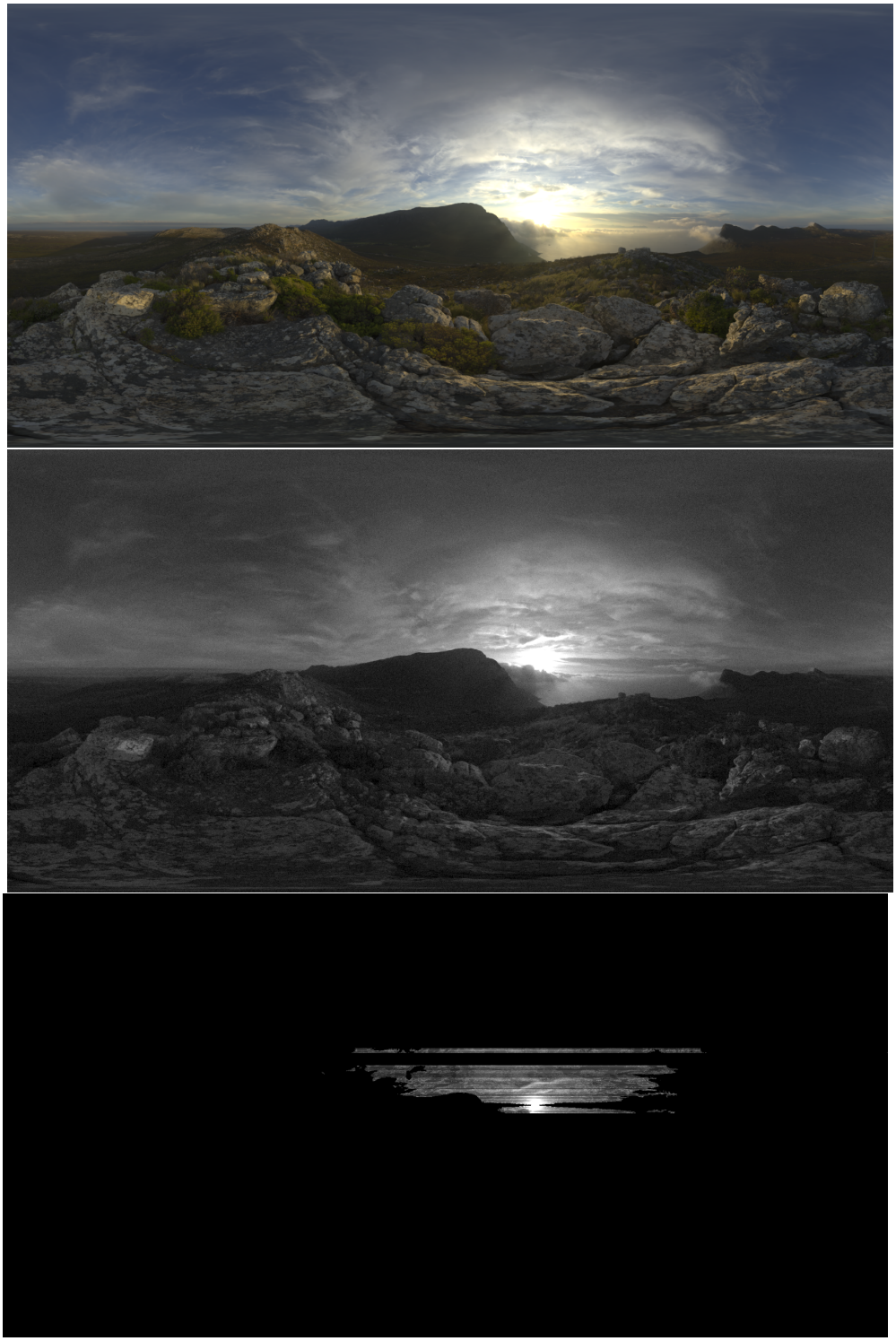

Figure 6.6: *Top*: An example of an environment light in the form of a tone-mapped HDR map. Courtesy of `hdrihaven.com`.
Middle: 100 million samples accumulated using the square histogram algorithm presented in section 6.3.1. Scale is not linear but instead the image was tone-mapped using the same algorithm as the HDR map. Our version, requiring two random numbers, was used, but the original fixed method generated the same image.
Bottom: Incorrect result caused by a typo inside the V_L as presented in [28].

Algorithm 3 Sampling from a discrete distribution

Input $p_i\ i \in [0 \dots n-1]$ probabilities
Output $V_i, K_i, W_i\ i \in [0 \dots n-1]$ the square histogram

procedure SQUARE HISTOGRAM
  $\alpha \leftarrow \frac{1}{n}$
  for $i \leftarrow 1 \dots n$ **do**
    $K_i \leftarrow i$
    $V_i \leftarrow \alpha.(i+1)$
    $W_i \leftarrow 0$
  end for
  for repeat $n-1$ times **do**
    $L \leftarrow argmin_i(p_i) \qquad H \leftarrow argmax_i(p_i)$
    $K_L \leftarrow H$
    $s \leftarrow \alpha - p_L$
    $V_L \leftarrow \alpha.L - s$
    $W_L \leftarrow \frac{-s}{\alpha}$
    $p_L \leftarrow \alpha \qquad p_H = p_H - s$
  end for
end procedure
procedure DRAWING SAMPLES
  $u \leftarrow uniform[0,1]$
  $j \leftarrow \lfloor u.n \rfloor$
  if $u < V_j$ **then return** j
  else return K_j
  end if
end procedure
procedure DRAWING SAMPLES (OURS)
  $u_1, u_2 \leftarrow uniform[0,1]$
  $j \leftarrow \lfloor u_1.n \rfloor$
  if $u_2 < V_j$ **then return** K_j
  else return j
  end if
end procedure

6.3.2 Multiple importance sampling

Multiple importance sampling is another generalization of Monte Carlo integration. Assume we want to estimate the integral

$$A = \int_{\Omega \subseteq \mathbb{R}^{\mathbb{N}}} f(x)dx$$

and we have two probability distribution $p(\omega), q(\omega)$. [2] provides an estimate of this integral as

$$\frac{1}{N_p}\sum_{i=1}^{N_p}\frac{w_{p,i}(x_{p,i})f(x_{p,i})}{p(x_{p,i})} + \frac{1}{N_q}\sum_{i=1}^{N_q}\frac{w_{p,j}(x_{q,i})f(x_{q,i})}{q(x_{q,i})}$$

where w are weighting functions which satisfy $\sum_{i=1}^{N_p} w_{p,i}(x) + \sum_{i=1}^{N_q} w_{q,i}(x) = 1$ for $\forall x \in \Omega$. Subscript under x denotes from which distribution was the sample generated. The paper provides a choice for these weights in eq. (6.11) and names them *balance heuristic* when $\beta = 1$, *power heuristic* otherwise. We adapted them to our particular case of the two distributions. Later it also proves that these weights are nearly optimal in terms of reducing the variance.

$$w_{a,i}(x) = \frac{N_a a(x_i)}{N_p p(x)^\beta + N_q q(x)^\beta} \tag{6.11}$$

The paper uses this technique for direct lighting from area lights. We will use it also for the environment light. Choosing direction according to a diffuse BRDF does not work well with small lights as they will not be hit very often because the rays are essentially random. On the other hand choosing rays with respect to lights will work very well. This will stop working for highly-specular BRDFs and big lights since in the majority of time the sampled direction will result in very low $f(x, \omega_i \rightarrow \omega_o)$ value. The MIS technique tries to eliminate both these bad scenarios by sampling both BRDF and lights exactly using the presented equations. After integration into the developed algorithm, it can render reasonably well the scene shown in fig. 6.7.

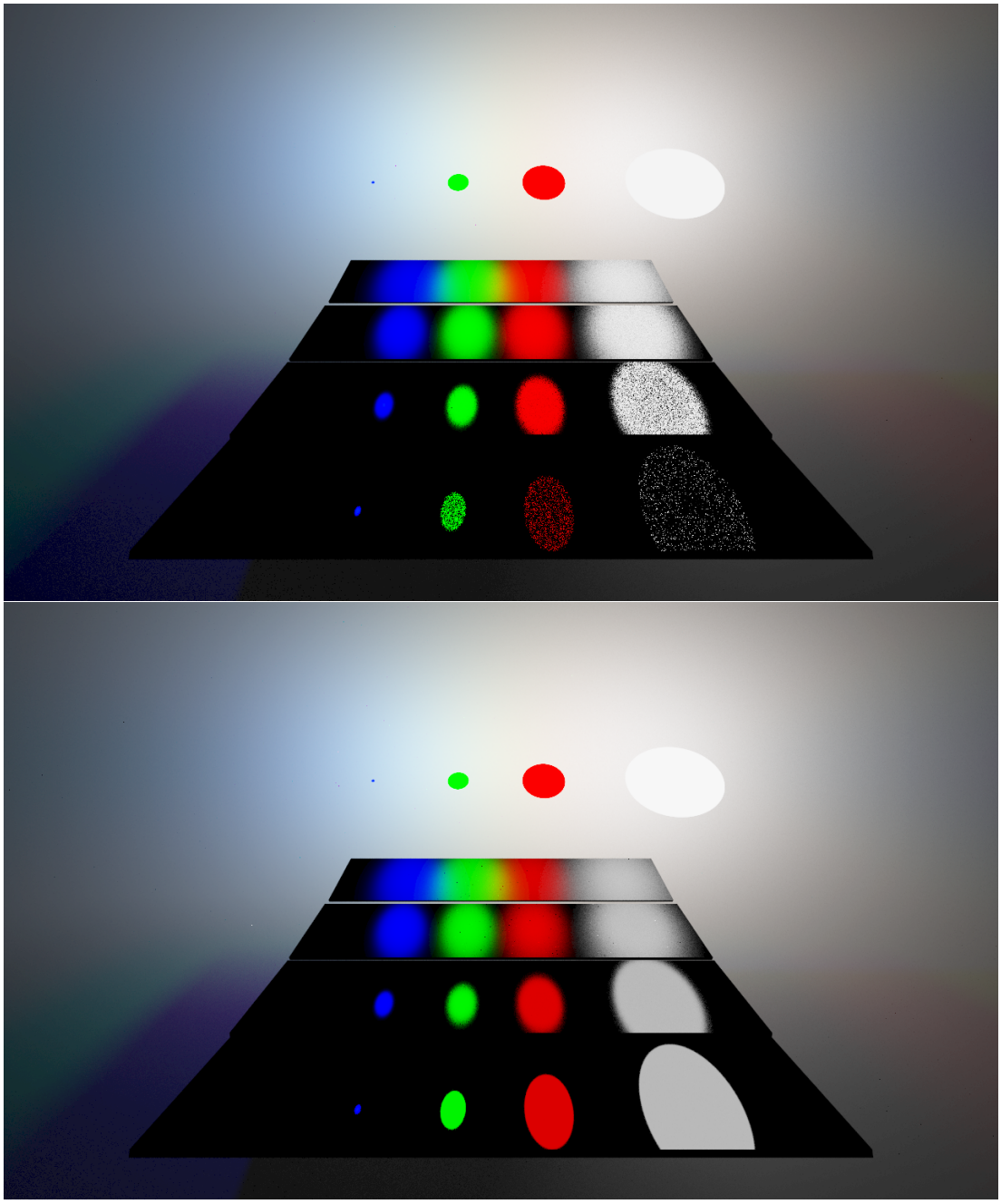

Figure 6.7: This figure is a recreation of Veach's MIS scene from [2]. The top image samples direct lighting according to the lights. The bottom one uses MIS for sampling using both the BRDF and light distributions. The light sampling fails for highly-specular BRDFs and big lights.

6.4 Final Path Tracing algorithm

Algorithm 4 Includes NEE and MIS.

$n \leftarrow 0$

$throughtput \leftarrow 1.0$ ▷ How much light can pass along the path
$LI \leftarrow 0$ ▷ Radiance arriving at the camera's pixel
$ray \leftarrow$ `GenerateCameraRay()`

while $n++ <$ MAX_PATH_LENGTH **do**
  $intersection \leftarrow$ `IntersectScene`(ray)
  if $intersection == None$ **then return** LI
  else if $intersection == Light$ **and** n==1 **then** ▷ Only cam. ray hit.
    ▷ Transfer the radiance via the path to the camera.
    return $LI + throughput * intersection.LI$
  else ▷ Hit an object
    $\omega_o \leftarrow -ray.dir$ ▷ Points away from the surf.
    $LI+=$ `directLightingMIS(intersection)`
    $\omega_i, pdf \leftarrow$`genNewDirectionAccordingToBRDF`()
    $costTheta \leftarrow \text{dot}(intersection.n, \omega_i)$
    $througput* =$ `BRDFEval`$(intersection, \omega_o, \omega_i) * cosTheta/pdf$
  end if
  $q \leftarrow \max(throughput.x, throughput.y, throughput.z)$
  if $q <$ `random()` **then**
    $throughput/ = q$
  else
    break
  end if
end while

7. Previous Work

In this chapter we introduce 3 main applications with the same goal that are very similar to each other. Then we formulate the requirements for our own solution.

7.1 BRDF dílna(workshop) - Master thesis

The master thesis[22] written by Jiří Matějka features a text editor for writing the reflectance functions which can then be applied to one mesh illuminated by an environment map. The user can define parameters for the BRDF that can then be tweaked during the rendering process. The graphical application uses the Qt library for the UI and mix of OpenGL&OpenCL for the image generation. It features progressive renderer and rotatable camera. The program however does not support area lights, although they can be simulated through the environment map, nor any indirect lighting. The user cannot input their own custom importance function, instead the author uses general method that tries to match the BRDF.

7.2 BRDFLab

This program [29] is capable of displaying wide range of BRDFs, including measured ones. The analytical model must be given in the form of a colection of *analytical lobes*, it does not seem to support arbitrary code. But the program does offer a fitting tool that can fit these lobes to any model. The BRDF can too be rendered using a simple model and an environment map. In addition, it supports 3D graphs with the plotted BRDF. The analytical models are written using special syntax in XML and GLSL, the rendering is done by Ogre 3Ds.

7.3 BRDF Explorer by Disney

This application offers similar features as the previous ones. It adds 2D graphs for the BRDF but does not feature custom importance sampling for the written functions either.

7.4 Our solution

Out solution should implement the presented path tracing algorithms and the user should be able to choose between its different variants; trading complexity for performance. Using a modular approach the previous direct-lighting-only solutions will be possible to implement without losing the developed tools. The concept of a scene allows more than one object to be created so the user can make a specific scene needed for their BRDF, including point and area lights. The scene can consist of objects presented in chapter 3 and is loadable from a file during runtime. This makes it easier for the user to make their own without the need to change the source code.

We want explicit support for custom importance sampling functions as they can have great effect on the quality of the rendered image. Same as the previous works, the program supports the notion of parameters and offers tools for working with them. The finished program should contain basic 2D graphs for inspecting the BRDF, its dependence on the parameters and the quality of the chosen importance sampling strategy.

8. Technologies

The described path-tracing algorithm throughout the previous chapters evaluates each pixel independently, that makes the algorithm easily pararellizible. This chapter introduces a few technologies for pararelization and then give the reasoning for the used ones in the application and describes these further in more detail.

Because of author's experience with C++ and OpenGL applications we decided to use them for our editor. They both offer high-performance and are not tied to a single platform or vendor.

8.1 Platforms For the Path Tracer Implementation

The following technologies were considered for implementing the path tracing algorithms. We only give a very brief introduction with what we consider are the advantages and disadvantages of each. The chosen technologies are more thoroughly explained later.

8.1.1 CUDA

is a computing platform that accelerates general computing by using NVIDIA graphics cards [30]. CUDA is a very mature platform with active community, plenty of available resources, and tools. C++ developer can directly write C++ functions that can be compiled for both CPU and GPU using `nvcc` compiler. In general, CUDA allows creating so-called `kernels` - functions that can be executed by the GPU. They are written in CUDA C++ and there is also an option for compiling these kernels during runtime. Meaning that C++ code can create a kernel from a string source code and then execute this compiled kernel on the GPU all during runtime without restarting the application. As was mentioned, CUDA and its tools are only available for NVIDIA GPUs.

8.1.2 OpenCL

OpenCL [31] is a general compute API specification currently developed by Khronos Group. In capabilities and usage it is very similar to CUDA. But the developer uses standard C++ compilers and OpenCL is accessed as a library through functions calls, all kernels are compiled at runtime from character strings. Great advantage over CUDA is that OpenCL is multi-platform. Hardware vendors are responsible for implementing the API for their devices and currently the OpenCL is available on multitude of devices including CPUs and GPUs.

Subjectively it has smaller community and fewer available resources. Although AMD offers some tools through its GPUOpen initiative [32] including an OpenCL profiler and debugger, they seem to only support AMD GPUs. Intel also offers its own tools[33] for developing the OpenCL applications on their integrated GPUs. To our knowledge NVIDIA offers no such tools. More importantly at the time of this writing, NVIDIA GTX 1050TI GPU with the newest drivers available still

only supports OpenCL 1.2 introduced in 2011 and this should be true for all NVIDIA GPUs. On the other hand Intel i7-7700HQ with integrated Intel HD 630 GPU both support OpenCL 2.1

OpenCL kernels are written in C99-like language and are compiled during runtime from strings. Since version 2.2 there is support for a subset of C++14 features[1] similar to what is now offered in CUDA. Access to OpenGL resources is available only as an extension, from experience it requires OpenGL and OpenCL to be executed on the same device.

8.1.3 OpenGL

OpenGL is a general graphics API specification also developed by Khronos Group. Although it is mainly targeted at raster graphics, since version 4.3 [34] there is support for general compute capability in the form of *compute shaders*. They are very similar to CUDA/OpenCL kernels but they do not offer explicit distinction between global and local/shared memory. Rather the user must rely on OpenGL textures and buffers. On the other hand sharing state between compute shaders and rendering pipeline of OpenGL is trivial.

8.1.4 C++

Although C++ does not have direct support for execution on GPUs, there is at least one library written around OpenCL that enable this - SYCL.

SYCL [35] is a C++ API specification, also created by Khronos Group, that brings a similar functionality of mixed CUDA, C++ to OpenCL world. Using SYCL user can call c++ SYCL functions that will be seamlessly executed by OpenCL. We encountered this library during very early stages of development but due to severe lack of resources and available implementations it was not researched further.

8.2 Chosen technologies

We decide to use OpenCL mainly due it being cross-platform library even capable of running on a CPU which would hopefully allow more thorough testing. The program was developed on a laptop with NVIDIA 1050TI GPU and Intel CPU with an integrated graphics card. Thus OpenCL can be easily tested on 3 different devices just by switching between them during runtime. The developed application supports this as it became useful during the development. Because undefined behaviour often manifested in different ways between the NVIDIA and Intel implementations.

[1] No dynamic allocation, exceptions or virtual functions.

8.3 OpenCL and OpenGL

8.3.1 OpenGL

OpenGL was mentioned because of the compute shaders. We used it to implement the visual part of the program.

Most of the OpenGL functions are only implemented as part of the graphic drivers and must be loaded at runtime. This can be accomplished by loading libraries, one being *glad* which can generate custom loader for different versions of OpenGL and its extensions.

OpenGL itself is not capable of creating the window context which is platform-dependent operation, nor it has access to any I/O. For this reason we use GLFW library which is cross-platform and works nicely with OpenGL. It offers callback-style API for processing I/O - in our case mouse and keyboard. After creating the window, glad is initialized using `gladInit`, then we can call any loaded OpenGL functions. The last library we used is `ImGui` [2], since GLFW does not do any custom UI drawing. This C++ library does not offer all features that e.g. Qt might but, personally I like its immediate style:

```
1  if(ImGui::Begin("HelloWindow")){
2      ImGui::Text("Hello, world %d", 123);
3      if (ImGui::Button("Save"))
4      {
5          // do stuff
6      }
7      ImGui::InputText("string", buf, buffLen);
8      ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
9  }
10 ImGui::End();
```

This will draw a window containing the text, input field, slider, and the button which will *do stuff* if it is pressed. Almost all the UI state is hidden and kept by the library. The example was taken from its official github page and slightly expanded, see more examples with pictures there. This style allows quick prototyping and a simple if statement can decide what is drawn. Great feature of this library is that it does not contain any OpenGL calls. All commands are deconstructed into simple geometric primitives and stored inside internal buffers, the developer can then processes these buffers as needed. We use OpenGL to draw the UI on the screen. This two-stage drawing will become useful later.

8.3.2 OpenCL

OpenCL is our choice for the implementation of the rendering algorithm, in particular its 1.2 version to ensure compatibility with NVIDIA GPUs. The API is in many ways similar to OpenGL. To use it one can choose and install one of the available frameworks and compile the project using the provided libraries. Since there might multiple such frameworks (platforms) installed on one machine, OpenCL is now usually available as a dynamic library (e.g. opencl32.dll on Windows) that calls the actual implementation for each platform. Because of this we decided to follow the same approach as OpenGL does through `glad`. But at the

[2] `https://github.com/ocornut/imgui`

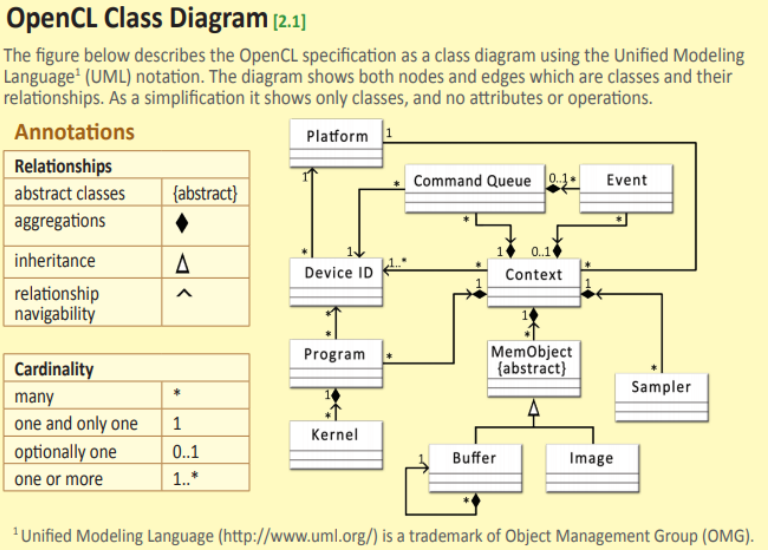

Figure 8.1: Shows core OpenCL concepts and their relationship to each other. Image taken from OpenCL API 1.2 Reference Guide[31]

start of development we did not find any *glad*-like library and so we did write our own since the official OpenCL (both C and C++) header files are freely available from the official Khronos Group Github[3]. Only the implementation must be loaded at the run-time. This allows the code to run on any machine with OpenCL implementation - usually part of the graphic drivers.

The OpenCL API defines a few key concepts. which are written below and their relationship is shown in fig. 8.1

1. **Platform** represents an OpenCL implementation which can contain multiple devices. E.g. a CPU and its integrated GPU.

2. **Device** is a hardware unit capable of doing the computation.

3. **Context** is a collection of devices from one platform. The context holds the state of these devices, including any allocated resources, kernels and queues.

4. **Command queue** is tied to one device and is used to send commands to this device. These commands are then executed asynchronously on the device.

5. **Program** is a source code compiled for a given device. It can contain multiple **kernels** which are functions that can be executed using the associated command queue.

Since OpenCL is intended for parallel execution, only executing one kernel once would not be very useful and instead hardware architectures are build for executions in huge batches. When a kernel is sent for execution on the command queue, number of *work items* and *work groups* must be specified. The work item

[3] `https://github.com/KhronosGroup`

corresponds to one call to the kernel's function. These work items can be clustered into work groups.

OpenCL supports up to 3-dimensional ranges of these items. For example if we have two matrices of $64x64$ elements and want to add them, then we might use a range of $(64, 64, 1)$ items and each item will be responsible for adding the two values. We could have launched the kernel with a range of $(4096, 1, 1)$ and do the indexing of the correct row, column manually. But that is unnecessary because this feature is intended to ease the indexing and thus a range the most suited to a given problem should be used. In our application we use 2D kernels to implement the rendering and 1D kernels to create graphs.

OpenCL distinguishes between *host device* (called host) memory and *compute device* memory(called device)[4]. Host device is the device calling OpenCL functions, in our case the CPU through C++ code. To dispatch any useful kernel, arguments are needed. Native types can be passed directly, but bigger objects must be transferred using buffers or images - shown in fig. 8.1. Buffers can be allocated both in the host and device memory but in the former case accessing such buffer from a kernel can be very slow.

Inside the kernel OpenCL distinguishes between 4 types of memory - global, local, constant and private.

1. **Global** memory refers to either host memory or kernel arguments which use the `global` specifier. Access to it might not be cached and latency is the highest of all three but its size can be in gigabytes for current GPUs and even higher for CPUs. With exception of images it can be read and written to, and is shared between all launched work items. But the access to it is not synchronized and memory barriers must be used to achieve that.

2. **Local** memory is shared between work items inside the same work group and different work groups their own. This memory can be really fast but its size is severely limited, OpenCL guarantees at least 16Kb each work group.

3. **Constant** is the same as global memory but the kernel cannot write to it and the size is limited.

4. **Private** memory is stored inside registers, GPUs have many of them in each compute unit. It is used to store all local variables.

The individual work items on GPU do not correspond to CPU threads. Instead a *warp* of 16,32 or 64 items is executed in lock-step. Meaning that all the items execute the same instructions on different data - SIMD. The warps are assigned to compute units - cores - to be executed, their assignment and context switching is up to the implementation. GPUs perform best if there are more warps than cores as they can swap-out the ones waiting for memory. The warps consist of work items from a same work-group which means that work groups should contain a warp-multiple of items and so it beneficial to set them appropriately even in cases where the local memory is not needed.

This lock-step mechanism may lead to *branch divergence* - a situation where items in the warp want to execute different instructions. This can easily happen

[4]OpenCL 2.0 added shared virtual memory but it will not addressed here.

when one group takes the `if` branch and the other the `else` branch of a conditional statement. The solution is to use masking, meaning both branches are executed by all work items and only those who are active will write the results to memory. This leads to decrease in performance since some items are idle, in the worst-case scenario only one item is active and the rest must wait. The longer the divergent code is the worse effect it will have. In our case, we launch the 2D kernel for each pixel, so the items in a warp process nearby pixels and generate coherent camera rays. But after the first bounce, the rays typically go into very different directions and may even hit different objects with different materials. This will lead to branch divergence, resolving this is not trivial and some papers propose different solutions [36][37].

8.3.3 Sharing Resources

In the brief introduction of technologies there was a short sentence about sharing resources between OpenGL and them. This is because the OpenGL is used for drawing and since it might very well run on the same device as the kernel will, it would be nice if one could directly draw the output of an kernel to the screen.

This is in some cases possible, OpenCL contains extension functions[5] that allow to share buffers and textures between OpenGL and OpenCL. Our program takes advantage of that if available, details are section 9.3.

[5]These functions are not guaranteed to exist on the target machine

9. Editor Architecture

This chapter provides a general overview of the program's architecture, interdependencies and interesting decisions taken during its development.

As the result is a graphical application, the user interacts with the program through a graphical interface. This must be reflected in its design and we considered the following as a typical user work-flow:

1. Write a new reflectance function.
2. Define its customizable parameters.
3. Optionally specify a distribution according to which should the BRDF be importance sampled.
4. Create a concrete example by setting the parameters.
5. Let the program plot the BRDF, see if it behaves as expected.
6. See the effect of the defined parameters on the function.
7. Use these graphs to sort out any implementation errors.
8. Prepare a basic test scene.
9. Render the scene and see how the material looks. This includes choosing the appropriate rendering algorithm.
10. Iterate the process by freely tweaking the parameters and lighting conditions.
11. Test effectiveness of different distributions and possible approximations both in the rendered scene and in graphs.
12. Save the developed functions for later use and also save the rendered image.

The developed program can be from a user's perspective partitioned into 3 main parts - *BRDF text editor*, *Scene Renderer* and *Graphs*. The application should, and does, freely allow the user to switch between them. A more detailed diagram is drawn in fig. 9.1. It summarizes which presented technologies were used for which parts of the program. Throughout this chapter we will describe some of these parts and the relationships between them in more detail.

We tried to create as modular system as possible, meaning that the user can switch between all the presented versions of the path tracing algorithm, as well as write a custom one. This allows implementing the one-bounce version used by the previous applications.

Before continuing further we show a few commented images of the final program serving as an introduction and quick demonstration of its features.

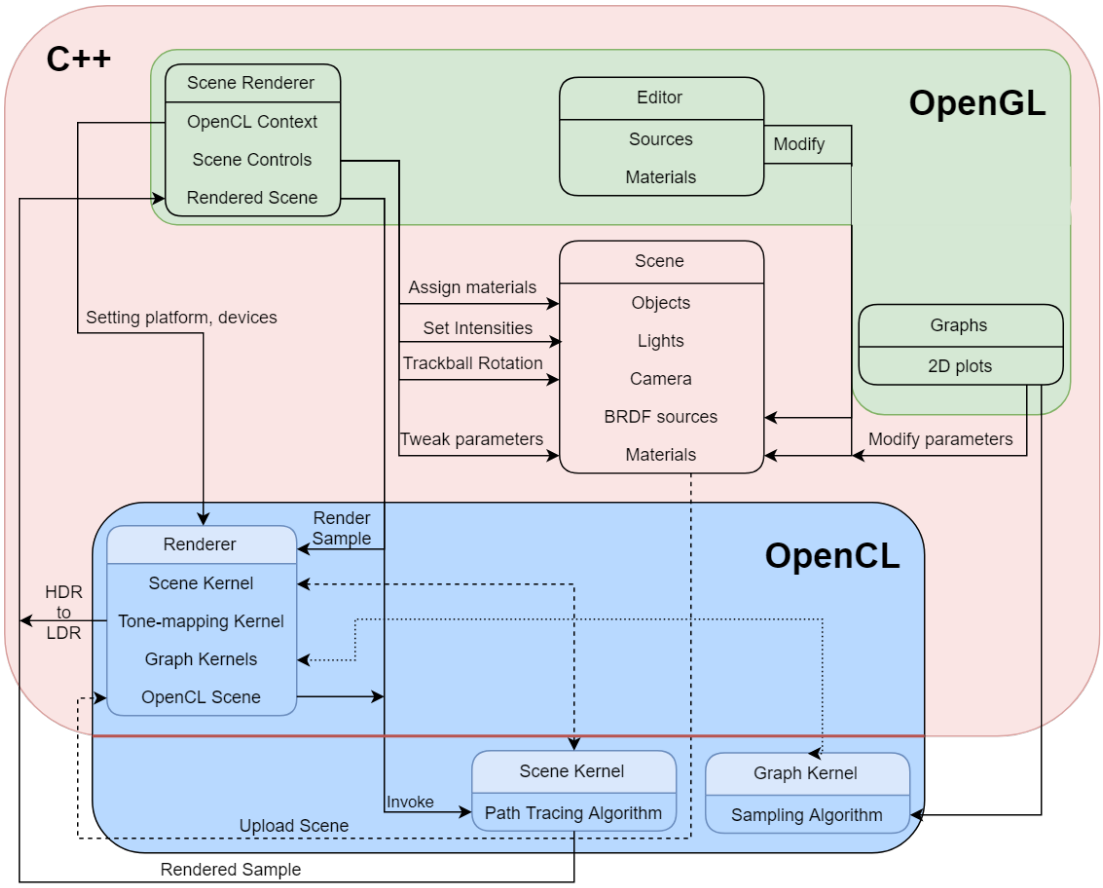

Figure 9.1: General overview of the program's architecture. It shows relevant dependencies between individual parts and which technologies were used for them.

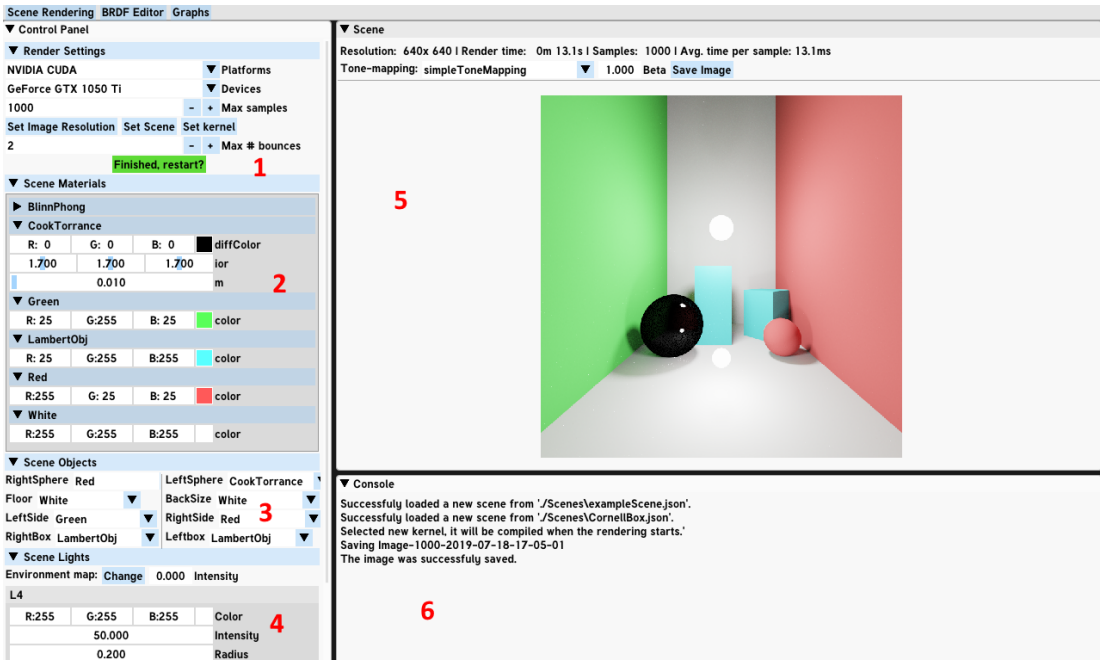

Figure 9.2: Scene rendering features of the program with the Stanford bunny.

1. Tools for choosing the OpenCL platform and device, including information about working interop with OpenGL. Offers buttons for loading a scene or selecting another rendering algorithm. Both are loaded from text files.

2. Contains a scrollable list of all materials' parameters where can they be modified. Clicking on a colour will open a small colour picker window.

3. Assigns materials to the scene objects.

4. Parameters for lights.

5. The rendered scene with some statistics, a user can currently select from two tone mapping algorithms. The scene can be rotated using left mouse button which acts as a trackball. This allows for some fine-tunning or e.g. inspecting a model from another side.

6. Console, that will show information regarding the materials and the kernel. The arrow in the top left corner can hide the console.

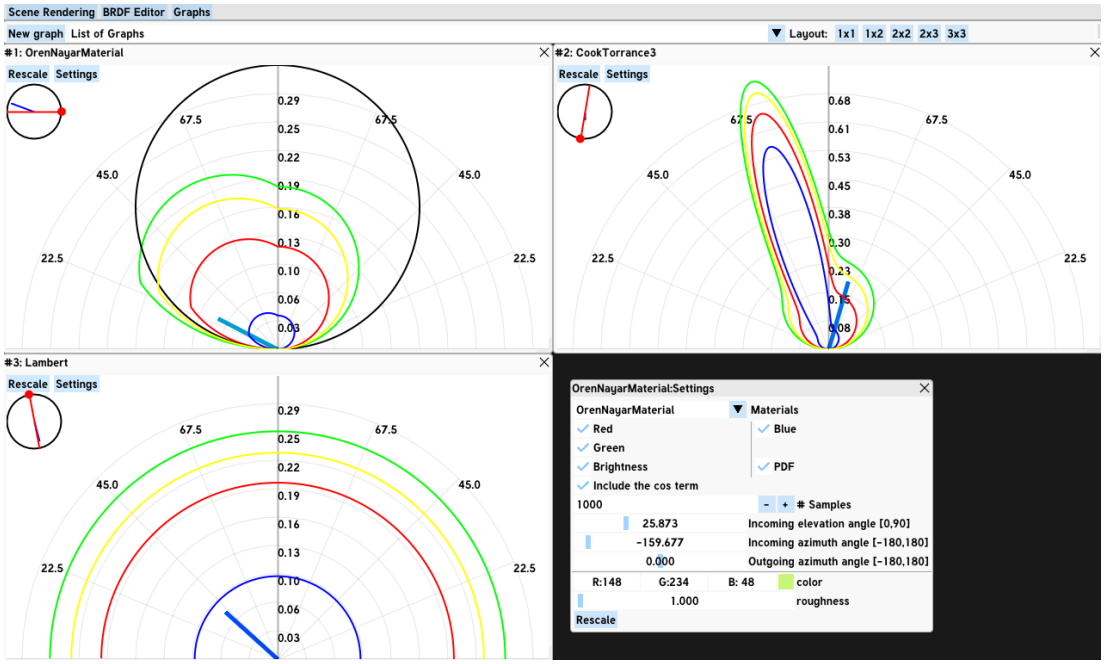

Figure 9.3: Graphs of 3 different materials, each graph has dedicated settings window that contains the material's parameters as well as precise controls for setting the incoming and outgoing directions. The graph contains values for RGB channels and calculated luminance (yellow). If a BRDF has custom sampling, it can plotted too - the black line in the top left graph. Optionally the cosine term can be included. The circles in the top left corners of the graphs allow setting the direction in a more visual manner using a left&right mouse buttons.

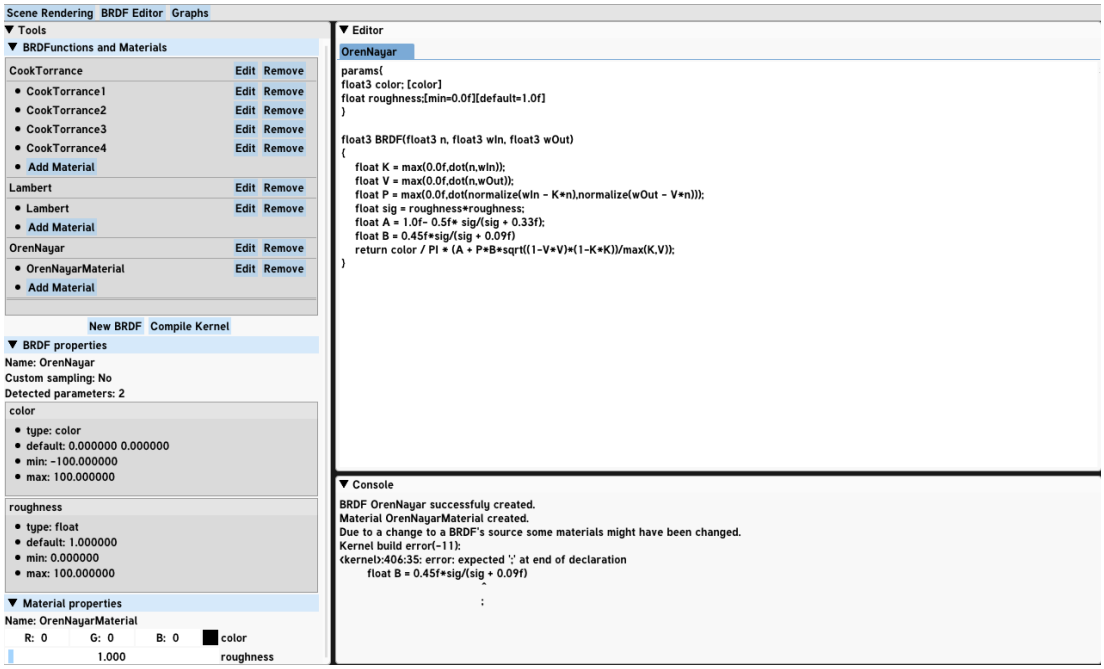

Figure 9.4: The editor features a small text source editor with multiple tabs. The left panel allows addition or removal of materials and BRDFs. The middle section shows parsed parameters for edited BRDF and in the bottom left corner there is a window with parameters for a selected material. The console will inform the user about any changes they made - e.g. adding a new BRDF, failure to add it or a kernel compile error message including the line number if possible. Unfortunately the NVIDIA compiler does not show correct line numbers but at least prints the line.

9.1 Program execution flow

As we mentioned, the user must be able to interact with the running program. In order to do so, we chose to use progressive rendering. The same approach was taken by all previous works and is also present in professional renderers e.g. Blender. It means that the rendering algorithm renders the image by adding one sample to each pixel at the time. This is reflected in the overall execution flow of the program, depicted on a diagram in fig. 9.5.

At the beginning of each frame, the user input from the last frame is collected. Next, if the user is in the Scene Renderer module and the rendering is enabled, then we dispatch the main kernel. As was explained in section 8.3, this runs asynchronously to the C++ code which is an advantage for us. In the meantime, we can process the necessary program logic solely related to GUI. One of the mentioned reasons why we chose 'ImGui' library is that it has two stages. Any code between `ImGui::BeginFrame()` and `ImGui::EndFrame()` is captured into command lists. Only when we have processed all the GUI logic, we wait for the kernel to finish and call `ImGui::Draw` which dispatches the stored commands to OpenGL. That means even though we use OpenGL to draw UI, we do not interrupt the running kernel with it.

9.2 Scene and Editor

We introduced objects that can be placed in the scene in chapter 3. Their storage is more or less straight-forward translation to C++ classes and OpenCL structs. A scene can be loaded from a .json file allowing the user to easily create new scenes, please refer to the user guide for an example.

We decided to make a distinction between a material and the BRDF. The latter is represented by its name and associated text source code that defines the BRDF with its parameters, described in the next section. The former is defined by a BRDF and concrete values of its parameters. Thus one BRDF can be used by more than one material and objects have an assigned material instead of just a BRDF and its values. We think this is a reasonable idea because it allows easier comparisons between two sets of parameters and at the same time easier sharing of them between two objects.

9.2.1 BRDF source code

OpenCL's syntax is almost equal to C's, we decided that users will be able to directly write OpenCL code that will be then used by the kernel. The similar approach is taken by all previous works as it is probably the simplest to implement.

In our case, the user must write one `BRDF` function with a fixed header. Optionally they can also define `SampleBRDF` and `GetwInPDF`. The former, in addition to the `BRDF`, chooses its own incoming direction ω_i to sample and also returns its probability. The `GetwInPDF` return a probability for a given direction, this explicit call is needed for the MIS technique. An example of the first two functions is shown fig. 9.6. Note that the user must either write only the `BRDF` function or all three of them.

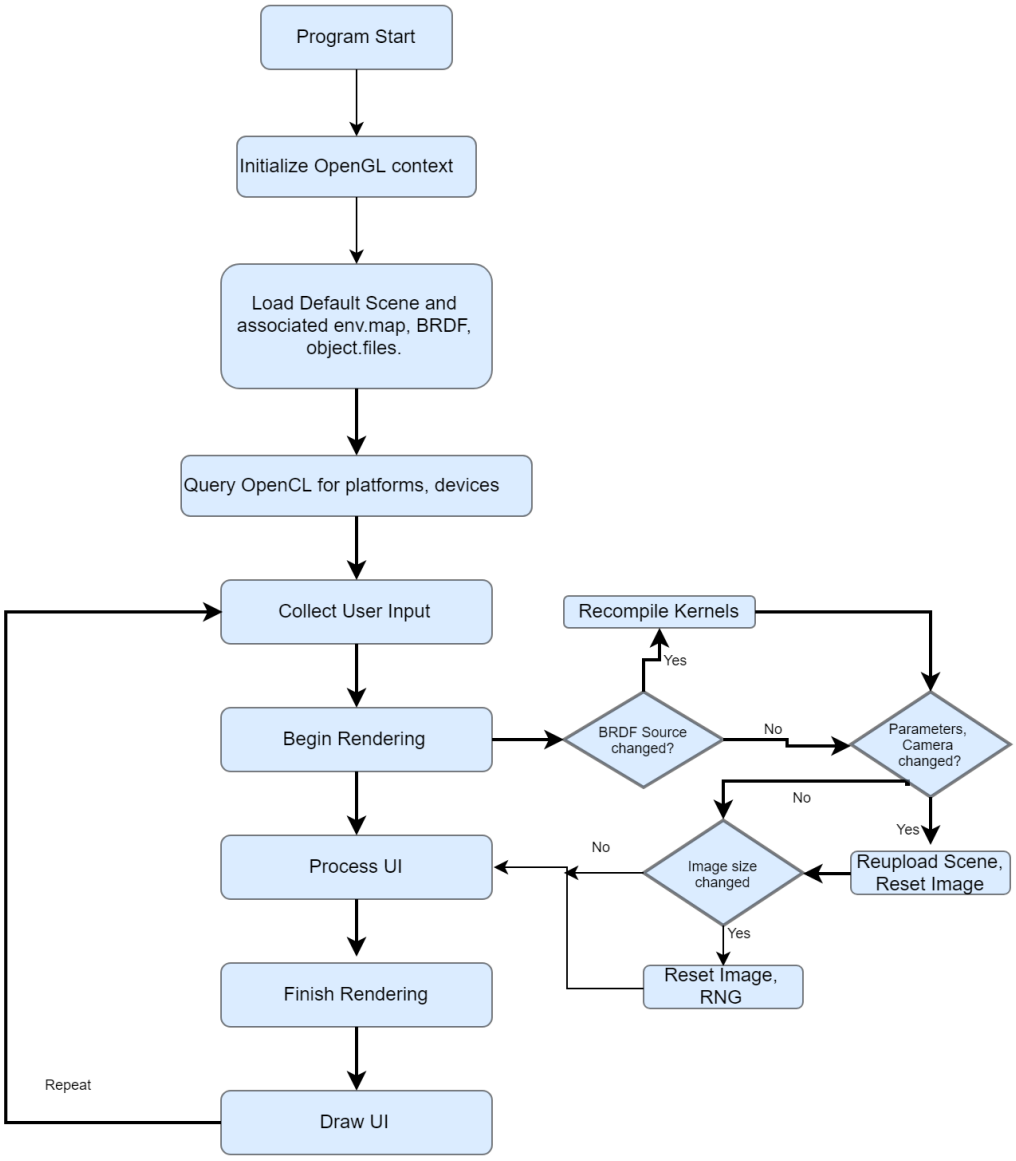

Figure 9.5: The execution flow of the program with detailed rules on the scene kernel recompilation.

We were considering making an explicit `BRDFDistribution` function that would accept an RNG generator and returned the sampled direction together with its probability. This would remove the need for `SampleBRDF`. That would also be a valid approach, but usually the reflectance functions share some code between sampling and evaluating, separate `BRDFDistribution` would either require a custom state structure to pass these computations to the `BRDF`, or there would be redundant evaluation. Also, some parts of the rendering algorithm, like NEE, do not need to sample a direction, they have both vectors and just need to evaluate the BRDF for them, thus the `BRDF` would still have to work without calling the `BRDFDistribution`.

On the other hand, our approach leads to code duplication between all three functions. This duplication can be moved to shared functions, but the user must pass the parameters manually and the name must be unique between all BRDFs. For the kernel compilation (section 9.2.4) purposes the user is not allowed to call `BRDF`, `SampleBRDF` `GetwInPDF` from any other functions.

We decided to keep all inputs in world coordinates. The reason is that most reflectance functions can be implemented without any goniometric functions because they can be replaced with dot products that are very fast in current GPUs. Nowhere in the rest of the path tracer we need local coordinates, so for the simplicity and efficiency a user must implement them themselves if they need them, or if they would make the code faster. That cannot always be done nicely which is discussed later.

BRDFs can have parameters, this is achieved by defining a separate code block called `params`. It must be written on the first line, an example in fig. 9.6. Currently allowed types are `int, float, float2, float3, float4`. The user can specify 4 tags: colour and minimal, maximal, and default values.

We adopted a similar technique as [22]. After defining these parameters, they can be freely referenced from either function.

9.2.2 Editor

One part of the program is very simple text editor that allows writing the BRDF sources and subsequently compiles the kernel. Picture of this editor is in fig. 9.4 and more pictures are shown in the user guide.

We did not try to make a best text editor with as many features as we could, years went into development of successful editors. Instead our editor offers basic text operations and allows loading new BRDFs from files as well as from the clipboard. This enables the user to use their favourite editor and copy-paste the code to our application when necessary.

This part of the program is capable of adding and removing new materials from the created reflectance functions. We added a simple overview of their parameters to see whether they were parsed correctly. Because of the dynamic kernel compilation there is a console that outputs the result of the recompilation and any compilation errors that might have happened due to user's mistake.

9.2.3 OpenCL scene

Due to OpenCL running possibly on another device with a separate memory, all the kernel's input arguments must be transferred to this device. We chose to use a `cl::Buffer` for each type of object presented in chapter 3. This leads to more kernel arguments but the user does not interact with them while writing the BRDFs, see appendix A.1 for the list of these arguments. An alternative approach would be to put all the types into one buffer of unions. The camera is passed as a separate argument.

Since OpenCL does not support a structure of buffers nor images, the environment map takes more than one kernel argument. In particular, two `Image2D` - one for RGB values and the second for the squared histograms of rows as they were presented in section 6.3.1. We can pack the probability, K and V arrays inside one `cl_float3`, this puts a limit on largest supported dimensions of 2^{23} in either direction as this is the largest integer representable by a 32-bit float. Another `image1D` of float3 is used for the marginal histogram.

Although the translation of a C++ structure to an OpenCL one is straightforward, care must be taken to ensure the correct alignment and padding. In particular, `float[3]` is not equal to `cl_float3` in C++ code nor to `float3` in the kernel. The specification[1] explicitly states that `float3` is equal to `float4` both in size and alignment. This is probably due to optimizations in hardware. For this reason we implemented the sphere data structure as one `float4` holding both position and the radius. We also explicitly use `float4` in type declarations instead of `float3` even if the fourth component remains unused.

9.2.4 Scene Kernel

Kernels were explained in section 8.3. In this section, we will take a closer look inside a scene kernel - the kernel responsible for rendering the scene. Its compilation process will be described together with some notable functions.

As we stated in OpenCL introduction - there are no function pointers, no dynamic memory allocation inside the kernel and neither can any global pointers be stored inside the buffers. This lack of function pointers can be somewhat alleviated by a switch statement on an integer. The restriction is based on hardware as not all supported devices(e.g. GPUs) must have a stack or `call` instructions. In that case, the compiler will inline all function calls in the kernel. This would of course not be possible to do with function pointers or recursion for that matter. Because our application features a general scene with more than one material and a ray can hit any of them, the manual switch has to be used.

Now, we will explain how the scene kernel is assembled. The process is demonstrated in fig. 9.6 on a small example. The assembly begins by fetching all materials and their BRDF sources from the scene. We collect all their parameters into one buffer `ParamBuff` containing `Param_t`s which are a union of all allowed parameter types. An offset is assigned to each material (not the BRDF) into this buffer representing its first parameter. Then we initialize the `Param_t`s to either the requested or implicit default values. An index is attached to each BRDF source and the and the written BRDF functions are prepended with the name

[1] `https://www.khronos.org/opencl/`

given to the BRDF. As was already stated, custom function are not renamed and their names should be chosen with care. In the end, we think these restrictions are reasonable and should not complicate writing any code too much. If the user does not choose to adhere to them, the worst-case scenario is a cryptic compile-time error message.

We still have to fix `BRDF`,`SampleBRDF`, `GetwInPDF` signatures by adding a couple of arguments - one to the buffer with parameters and one for the material offset. For this reason, those three functions cannot be called from the code. Then, the parameters' definitions can be pasted inside both functions and initialized with the correct union member of `Param_t` value obtained from the buffer based on the offset and type of the parameter.

This is a very flexible approach that can handle an unlimited number of read-only parameters of allowed types. Repeated access to them should be comparable to any local variable as they are usually stored in registers. Currently, arbitrary buffers are not allowed. One way of implementing them would be to not dereference the index position inside the buffer and leave arbitrary space inside before the next parameter. The reasons for not implementing them were that access to this buffer could be very slow on GPUs if the global memory is not cached. Also, the allowed parameters already cover a great deal of BRDFs models. One thing the buffer would add is the demand for precomputed tables or even numerical BRDF models. In that case, perhaps using an OpenCL *clImage* would be more appropriate but the number of image arguments passed to the kernel is limited so `Image3D` would have to be used to pack these additional arguments, due to complexity needed it was not implemented and it is an item in the future work list.

At last, general `BRDFEval`,`BRDFSampleEval`, `GetwInEval` functions are made, the first two can be seen in fig. 9.6. These are meant to be called from the rendering algorithm whenever a surface evaluation is needed. They accept the arguments necessary for calling the BRDF implementation, that includes the buffer with parameters and a `MatInfo` structure. This simple structure is the OpenCL equivalent of material and holds the index of a BRDF and the material's offset. All these functions contain a switch statement that is based on the BRDF's index and the code just calls the corresponding renamed BRDF with appropriately shifted buffer. Acting precisely like a function pointer.

`BRDFSampleEval` calls `SampleBRDF` if it was defined by the user. If not, then the `genCosineDir` (using cosine-weighted strategy) is called first, followed by a call to the `BRDFEval`. If `GetwInPDF` was not defined either, the cosine-weighted sampling is used too. Meaning that the rendering algorithm can transparently call all these functions depending on its needs. E.g. the `BRDFEval` is called by the next event estimation.

Had we asked the user to optionally define the `BRDFDistribution` instead of `SampleBRDFEval`, then this would be the place where the state structure (section 9.2.1) obtained from the `BRDFDistribution` would be passed along to the `BRDFEval`.

`MatInfo` is a part of `HitInfo` OpenCL structure generated by intersection tests between rays and the scene, see the implemented path tracing algorithm for precise details.

After these transformations, the main rendering algorithm is simply appended

to those two generated functions and compiled. Thus a custom algorithm can be easily plugged into the application as long as it has correct arguments - appendix A.1.

Again, the whole generating process is summarized by the diagram in fig. 9.6. The online compilation is thus a crucial feature needed for this editor. This (re)compilation is noticeable in the program, but does not take more than a few seconds. The kernel is only recompiled when the user explicitly requests it, or tries to render the scene after some source changes. The precise rules when this happens are discussed in the next section and displayed in fig. 9.5.

9.3 Renderer and Kernels

The renderer is the core unit of the program. It is an abstraction over the whole rendering algorithm. User can choose OpenCL platform, device, and the path tracing algorithm. Its goal is to separate rendering implementation details from the rest of the program. It contains the Kernel class which is another abstraction over a compute kernel. This way the OpenCL implementation is contained in only one place. Changing from OpenCL to e.g. CUDA would require rewriting these two classes and replacing any OpenCL specific UI controls. Of course, all written reflectance functions would have to be modified too. Also, objects in the scene can serialize themselves into corresponding OpenCL structures, likely the same structure could be used for other technologies. To summarize, the application is not that modular but the care was taken to contain this implementation-specific behaviour.

The renderer class is responsible for triggering the explained compilation process and dispatching the kernel into the command queue. The output from the renderer should be one added sample to each pixel. We did not discuss yet where are these pixels stored, how are they displayed, and how is the kernel actually called, this section will amend that.

We already revealed a few kernel arguments in section 9.2.3, appendix A.1 contains the declaration of the rendering algorithm kernel function with all its arguments. So it has all the mentioned ones and three new ones - `numSamples`, `buffImage`, and `rngState`. We will present them shortly by describing the overall rendering process. There is also a `maxBounces` algorithm corresponding to maximum length of paths.

The GUI (OpenGL) renders the scene as a 2D low-dynamic range (LDR) texture at the end of each frame as is depicted in fig. 9.1 and indirectly in fig. 9.5. We know that the kernel is called inside *Begin Rendering* part, so one idea would be to let kernel directly write to this texture. *Finish Rendering* then waits for this kernel to finish and OpenGL is then free to show this texture to the user. There a few reasons why we do not do it exactly like that.

First, it is not possible. We did mention that sharing objects between OpenGL and OpenCL is sometimes possible. Yes, it is if they both run on the same platform/device and the device supports it. We will discuss both cases shortly. But the main limitation is that the OpenCL images are not read-write, they can only do one type of the operation in the kernel. We could share buffers instead of images between OpenGL, that would work, but we do not need to do that.

The scene kernel has its own separate buffer argument `buffImage` and the

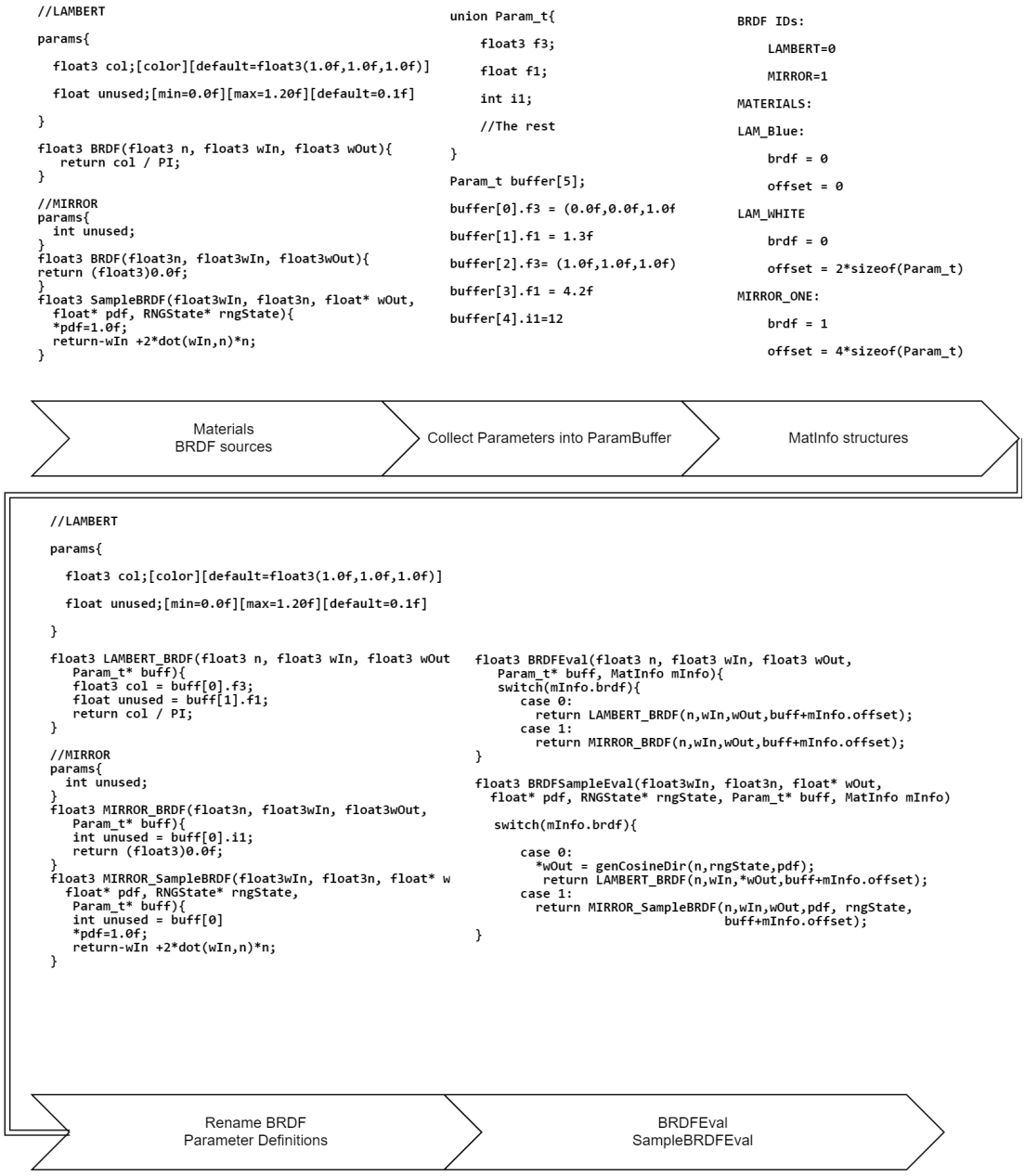

Figure 9.6: Assembly of the scene kernel. In particular this shows how are individual sources transformed, including their parameters. The rest is just appending the loaded rendering algorithm. We did not show the `GetwInPDF` for brevity as it is almost identical to how `SampleBRDF` is processed. See appendix A.1 for it's declaration.

execution between *Begin* and *End* will add one sample for each pixel to this buffer. Next part of the `End Rendering` is a tone-mapping kernel. Since the path tracing algorithm returns the radiance which can be arbitrarily large, the tone-mapping kernel is responsible for mapping these outputs to [0,1] range that can be rendered to the screen. Although both OpenCL and OpenGL support, and we use, the floating-point textures, they can still only render the [0,1] range.

This new kernel accepts the `buffImage` as its input and is responsible for writing the rendered scene to the LDR texture that can be used by OpenGL. It does this by first, averaging the sum in the buffer by the number of rendered samples. We could have done this inside the main kernel. Then a tone-mapping algorithm transforms these values to $[0, 1]$ range and writes them to the texture. We implemented two algorithms

$$map_1(v) = \frac{v}{\beta + v} \tag{9.1}$$

$$map_2(v) = \alpha v^{\gamma} \qquad \gamma \in (0, 1). \tag{9.2}$$

Both are very simple global filters, the gamma parameter does not map the range exactly to $[0, 1]$ and might leave some parts of the image under- or overexposed. Although the user cannot currently add new one, it only requires adding the code one source file and registering the added function. Only after this kernel finishes the control goes to *Draw UI*. In fact, we dispatch this kernel right after the main one, because OpenCL's command queue ensures it gets executed only after the first one.

This tone-mapping kernel still requires a texture. If the sharing with OpenCL is enabled, we directly pass the one used in OpenGL. We can do that because ImGui is not using any OpenGL functionalities in *Process UI*. If not, we create an OpenCL image, pass it to the kernel and then map both the OpenGL texture and the image to CPU's memory and do the copying there. This is, of course, slow and generates non-trivial overhead. For this case, we added the sharing capabilities to the OpenCL platform and device selection.

During the development we encountered a problem where the Intel graphics drivers are reporting that they support the sharing, but they actually do not and the application crashes as the result. Since this crash happens inside the graphics driver, we cannot do anything about it. The only recommendation we can give is to test all the devices and see which work and which do not.

`numSamples` argument signals how many samples are present in the `buffImage`, after some refactoring, it's not currently needed inside the main kernel, but is still present there if needed in the rendering algorithm. It's incremented before each *Begin Rendering*.

9.3.1 RNG state

In the theory, we introduced two pseudo-random number generators. They differ from e.g. Mersenne twister by having a very small state. We can use that and have an RNG for each pixel (`rngState`) i.e. each OpenCL thread as another argument to the kernel. Thus there is no need for any locking or shared memory between the threads. We have provided a simple `uniformRNG(rngState)` function accepting an opaque state, the state is available in `SampleBRDF` function which can be seen fig. 9.6. The user can call this function as many times

as needed. Every pseudo-random generator needs a seed that uniquely identifies the whole sequence. We can initialize, or reinitialize, all RNGs by writing to the `rngState` buffer. In the implementation we use C++11 `<random>` library that provides `std::random_device` which is capable of generating truly random numbers on the CPU. Since this is done only when needed we do not have to rely on fast generators and instead, we can ensure the quality of randomness.

9.3.2 Restart rules

Multiple user actions can trigger a change that should restart the rendering process from the first sample or even recompile the kernels.

First, if the kernel is not compiled at all, or is out of date because a BRDF got changed, it is recompiled. Second, if anything in the scene changed, we must update the corresponding buffers. This includes any parameters, camera or light intensities. Of course, this should also restart the rendering process. We do that by setting `numSamples` to 0. The rendering algorithm is written in such a way that it overwrites the `buffImage` in this case instead of adding to it. We do not have to, and we do not, reset the `rngBuffer` with new seeds. It's not necessary and would incur noticeable lag as the used `std::random_device` is not fast enough. Next, if the texture size changes, we have to restart the rendering and recreate new image buffers, this time including `rngBuffer`. Lastly, if a device is changed, or even the platform, we have to stop the rendering process. We simply destroy the OpenCL context associated with this device and later create a new one. This destroys all compiled kernels and stored buffers.

Overall these rules are summarized in a digram in fig. 9.5. Due to relatively small scenes and a low number of total parameters, rotating the scene and changing the parameters is nearly instantaneous and the scene is just *"locked"* at one sample per pixel during these changes.

9.3.3 Custom Kernels

The renderer has also the ability to generate custom kernels where the programmer supplies the function as a string and can execute them whenever needed. We use this functionality to implement fast sampling of reflectance functions for Graphs. We "issue" these kernels through the renderer, because they are invalidated when the OpenCL device is changed and must be recompiled when the source is altered.

9.3.4 Implementation shortcomings

We tried to give the reasoning for our choices in designing this renderer architecture. Yet there are some unresolved issues. We heavily rely on the user cooperating when implementing the reflectance functions. We believe we have set reasonable expectations but one can easily break the kernel by e.g. defining new global functions. For this reason, we provided the console, which prints any kernel compilation errors.

But this process relies on the vendor's compiler to deliver these errors. Since we piece different source code together, the shown errors might not be detected

at the expected places, especially if the errors are in the global namespace. The reported line numbers are not always correct either. We tried to at least partially solve this problem by using `#line X` macro present in C99 and also in the OpenCL specification. But the current NVIDIA drivers seem to ignore it, so an error on a first-line in e.g. Lambert's 5-line BRDF might show up on the 500th simply because there is a long, complicated BRDF implementation above it.

One issue [22] dealt with was the limit of the longevity of kernel execution. Running too long and the graphics card gets restarted. We did not encounter this problem during ordinary work with the application. But the limit still exists, and running very complicated kernel with a complex scene can certainly hit it. In that case, splitting the scene into blocks as [22] did will help, but it is not currently implemented.

9.4 Graphs

The last discussed part of the application is its plotting capabilities. Our application supports multiple graphs as it can be seen e.g. in fig. 9.3. This is partly due to 4 plotted lines per graph and so it is not very practical to try to put multiple materials inside one graph. Instead, an option for more separate graphs was added as it might be beneficial to compare different materials between each other. The windows are freely resizeable and hideable.

One graph shows a 2D slice of a given reflectance function, or more precisely a material, in the plane of outgoing ray ω_o and the normal. All three channels can be shown together with the luminance[2]. The plotted curves are constructed by evenly sampling the outgoing elevation angle. Optionally the user can enable multiplication by the term $n.\omega_o$, which then more accurately represents the actually reflected light from the surface.

The user can use the controls located in the top-left corner to rotate this reflection plane around together with the incoming direction. For more finer controls there are also sliders with angles that can be used, see fig. 9.3. Another feature is direct access to the material's parameters that can be freely changed in the same way as in the rendered scene.

9.4.1 Computing graphs

In order to create the plotted lines, the material is evaluated with different ω_o values sampled evenly with respect to the elevation angle. Since this can be done in parallel, naturally, a kernel is used.

This kernel is also build dynamically similarly to the main one. Since only one material is present and ω_o is known, there is no need for creation of `BRDFEval` nor `SampleBRDFEval`,`GetwInPDFEval` functions. Instead the BRDF source can be directly pasted inside the kernel.

[2]Computed as $0.2126 * R + 0.7152 * G + 0.0722f * B$ as per [38]

9.5 Implemented algorithms and their performance

The application is accompanied by 3 path tracing algorithms.

- `pathTracer`: The basic version of the path tracing algorithm as presented in section 2.4 and used in fig. 2.1.
- `pathTracer-DL`: This algorithm features NEE by directly sampling lights, does not use BRDF sampling, its output can be see in fig. 6.1.
- `pathTracer-DLMIS`: Uses the MIS technique and samples the new directions using BRDF custom sampling (`BRDFSampleEval`). Was used to generate images in fig. 6.7 and fig. 6.2.

We measured their performance on a NVIDIA 1050TI GPU, the testing resolution was 640x640 pixels and the times represent the amount of time spent inside the rendering kernel per sampled calculated as average from 500 samples. Three scenes were used for comparing the performance of these algorithms: The Cornell box-like scene already used for showing the algorithms, an outdoor scene containing the Stanford bunny (fig. 9.7), and the default scene (fig. 9.8). The results are shown in the following table:

Algorithm [ms/sample]	#max bounces	Only Bunny	Cornell box	Default
`pathTracer`	1*	3.1	1.0	1.0
	2	3.9	2.1	1.9
	5	4.0	6.0	3.0
`pathTracer-DL`	1	6.5	7.0	3.1
	2	6.9	13.3	4.0
	5	7.0	23.9	4.8
`pathTracer-DL-IS`	1	7.1	11.2	6.9
	2	8.0	27.1	11.0
	5	8.3	67.3	21.6
`No shadows`	1	4.1	4.1	3.0
	2**	4.1	11.2	4.0
	5**	-	-	-

The fourth algorithm is `pathTracer-DL-IS` with disabled shadows. Meaning that lights are sampled without taking visibility into account. This matches as close as possible the output of the master thesis[22]. *Only one bounce for the basic path tracer means that all objects are black and only the lights directly visible from the camera are drawn. **Due to absence of shadows the lighting in this image is incorrect and thus was not measured. But if we understand the thesis correctly, the OpenGL approach was able to generate one sample under less than a millisecond in some cases. And that is on hardware from 2010 but we also chose more complex mesh. None the less

The results show that simple scenes can indeed be rendered interactively even with indirect lighting. The worst result is for the path tracer with MIS, the biggest portion of that time is taken by intersections of shadow rays with the scene . But 67ms per sample still allows to work with the application and even

Figure 9.7: This scene features the Stanford bunny illuminated by the environment map. Uses the Cook-Torrance material with grey diffuse color and roughness of 0.24 .

change the parameters interactively. Furthermore, MIS should converge faster in some cases.

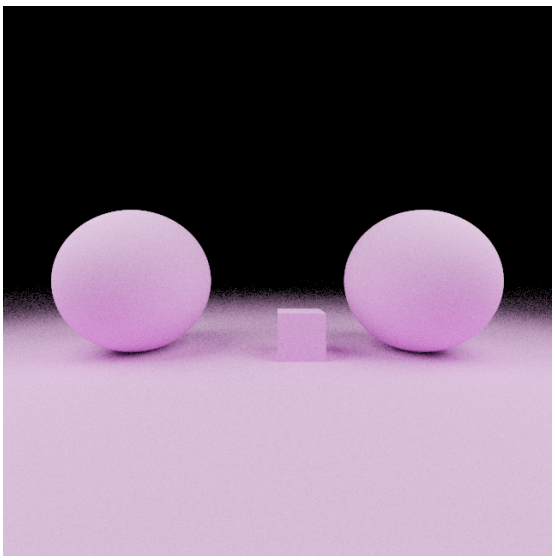

Figure 9.8: This is the default scene loaded into the program. Contains two spheres and a box on a horizontal plane. The scene is illuminated by two area lights from sides (not visible).

Conclusion and Future Work

First, we introduced the necessary theory to derive the well-known path tracing algorithm, explained the scene and its objects, described a few reflectance functions. After that, we explained the techniques which improve the convergence speed of the rendering algorithm and incorporated them into our project. section 7.4 reviewed the previous work and proposed a new solution, the following chapter 9 stated the expected usage and explained the developed program with its interesting concepts.

We believe that we have created an editor that is comparable with the previous works and offers features that they do not. Simultaneously is modular enough to match the users' needs. Thanks to the usage of OpenCL technology, we achieved interactive rendering times for simple scenes with a few objects and indirect lighting. As a proof of its capabilities, almost all images used in this work are created using the program.

Part of the work is the implementation of the path tracing algorithm and its features as specified in the theory, presented reflectance functions and a couple of scenes for testing purposes.

9.6 Future work

Although the editor is working, there are few needed improvements or non-ideal parts. The most severe is lack of consistent tangent vectors for meshes. The user can use `genBasis` function to generate the tangent vectors but they are not consistent across the whole mesh. This was discovered too late to properly address. Another unsupported thing is specifying default parameters' values in the scene file. Because now, the scene loading just uses default parameters assigned by the BRDF, this leads to e.g. having 2 Lambert materials in the scene file which will then be initialized to the same default value even though the first thing the user will do is differentiate their parameters. Although this does not impact the rendering, it does impact the productivity.

There is certainly much room for kernel optimizations, in particular the local memory could be used to store parts of the scene. The master thesis[22] uses OpenGL to generate camera rays and achieves much faster rendering times than our fastest kernel. This idea could be leveraged for our application too, OpenCL would then handle indirect lighting. But this creates additional complexity because the whole scene would have to be decomposed into triangles.

Future work can be done by implementing more complex rendering algorithms, bidirectional path tracing would add support for caustics that can be generated by some reflections. Adding new object types to the scene is always an option. As we discussed, the BRDF parameters can be expanded by adding a general buffer parameter.

We did not implement 3D graphs but they could be very valuable to inspecting the BRDF and even its sampling function.

An interesting feature would be to in some way export the developed reflectance function. This could either mean evaluation at predetermined angles, making this a measured BRDF. Or export in some general format supported by a

commercial renderer. Blender allows scripting materials using python and building them using nodes from basic building blocks. The latter would also be an interesting feature.

Currently, the default sampling strategy for unknown BRDF is to only use the cosine term. [22] used another method that could approximate the distribution based on the BRDF. In this way, our application could too be improved. Or at least the BRDF could be discretized and then the square histogram method can be used as the sampling strategy.

Personally, I would like to explore the space of BRDFs defined in [19] and see which BRDF can be generated from it.

10. User Guide

This goal is of this document is to show how to obtain, deploy and work with the developed application.

10.1 Obtaining the application

The developed application is distributed as an electronic attachment with this thesis. These attachments are:

1. Source files used to build the application.
2. Pre-built binary executable `BRDFEditor.exe`
3. Supplementary materials which include the written kernels, scenes, HDR maps and the BRDFs used in this thesis.

10.1.1 Compiling this project

Since there are pre-compiled binaries available, this step can be skipped. The program uses CMake to generate the project files and is known to work with CMake 10.2. Although all the used libraries are cross-platform, the compilation was not tested on Linux nor MAC, the safe way to successfully compile the project on Windows is to use Microsoft Visual Studio. First, generate the project files using these commands:

```
1    cd <Directory with the "BRDF Editor" folder.>
2    mkdir build
3    cd build/
4    cmake "../BRDF Editor"
```

If the Visual studio is properly installed, CMake will generate the `.sln` project that can be built, preferably in the Release configuration. There is also another way, Visual Studio 2017 and newer support loading the cmake projects directly. It requires installation of "Visual C++ tools for CMake" - please see the Visual Studio documentation[1] . Then the `.sln` project is generated on-the-fly and can be build the same way.

10.1.2 Running the application

The application requires OpenGL 4.5 and installed OpenCL 1.2+. If these are not present, the application will not launch and will report an error. Please make sure that both `opengl32.dll` and `opencl.dll` are available. Both should be installed together with the graphics drivers. There should be no need for any SDK like [33] but installing them will not cause any harm.

The program window is resizeable but the intended resolution is somewhere between 1280x768 and 1920x1080. Smaller resolutions may result in clipped UI

[1] At the time of writing, the page is `https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=vs-2017`.

elements, there is not much benefit for larger screen expect for graphs. The application runs on 4K screens but the font does not scale, this is a limitation of `ImGui` library.

On the start, the default scene and kernels are loaded. These are `pathTracer.cl` and `defaultScene.json`. Now, we will show the usage of the program through the following commented images.

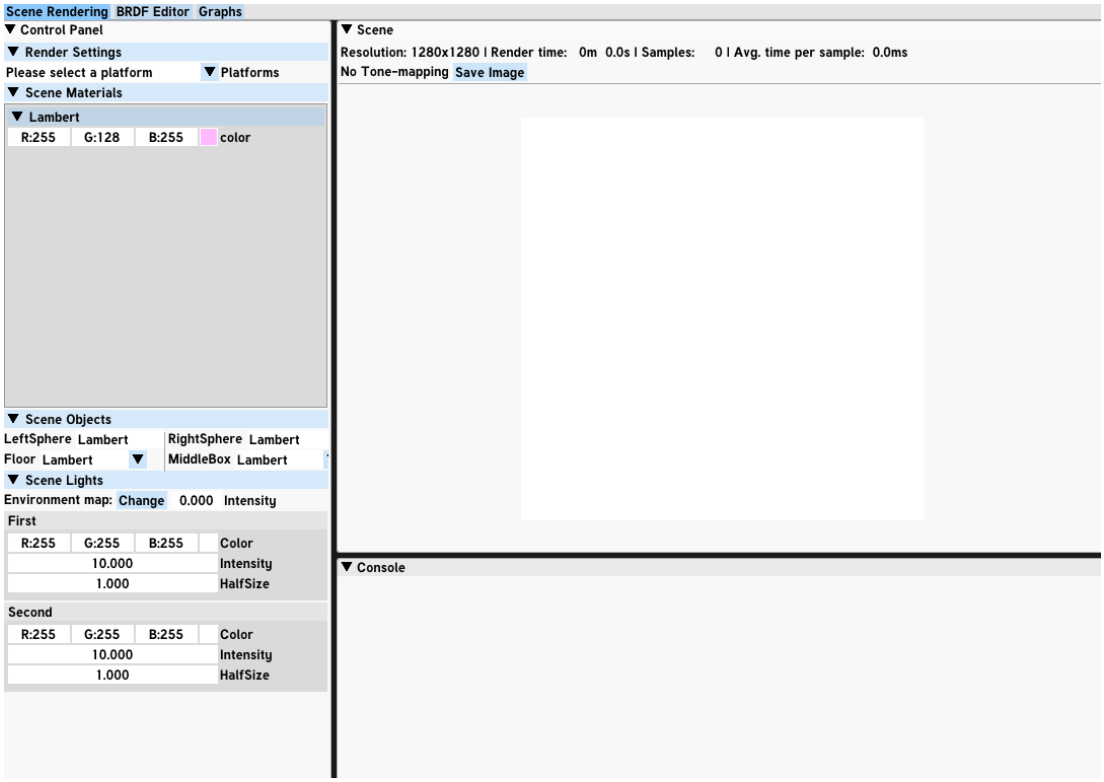

Figure 10.1: The program after launch should look like this. First thing we need to do is to select a OpenCL Platform and a device.

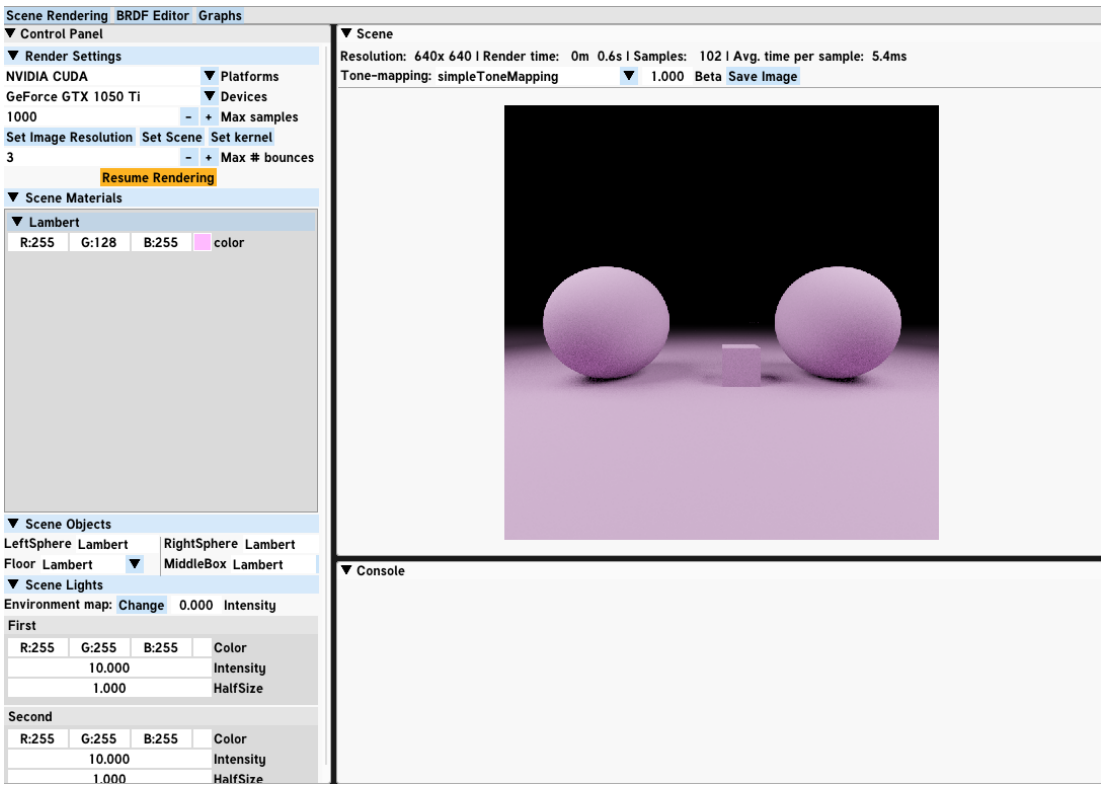

Figure 10.2: We have selected the NVIDIA platform and its GPU. After this, a few new buttons appeared. Now the user can freely change the scene, algorithm and the image resolution. We have set ours to 640x640 and hit the *Start Rendering* button. The rendered image begins to appear. We can move the camera around by dragging the left mouse button across the image. Leaving the scene aside for the moment, we pause the rendering process and move to "BRDF Editor" window.

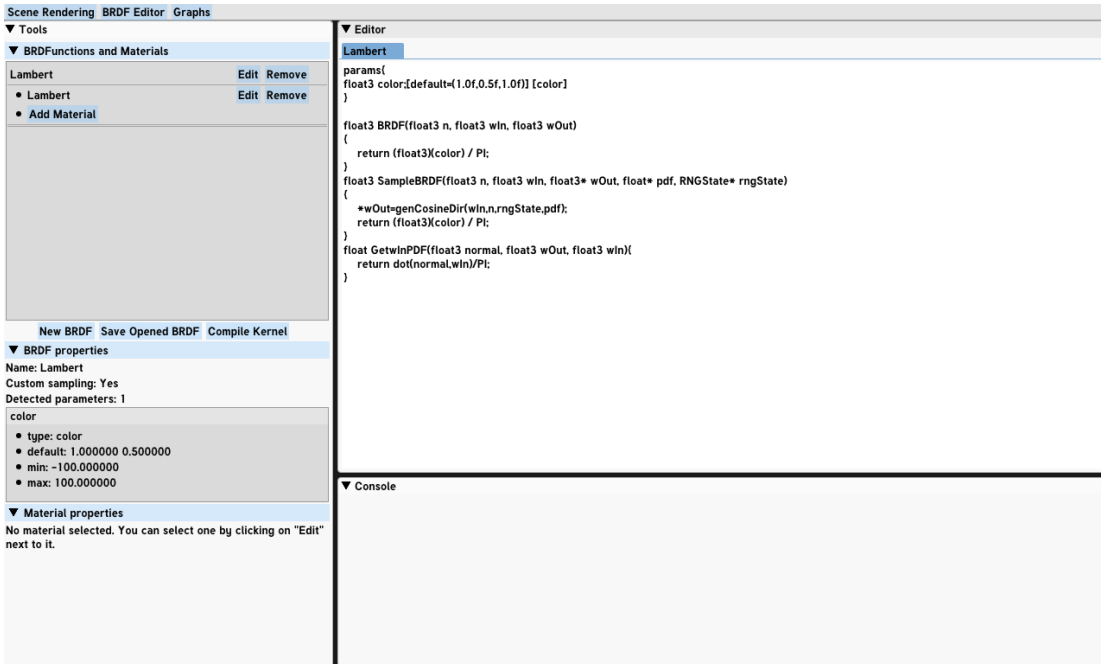

Figure 10.3: The program now switched to its editor, there is already one BRDF with a material present - the pink Lambert material. Clicking on the *Edit* button next to the BRDF will open the editor. Here we can see the implementation of the Lambert BRDF, including custom sampling and parameters.

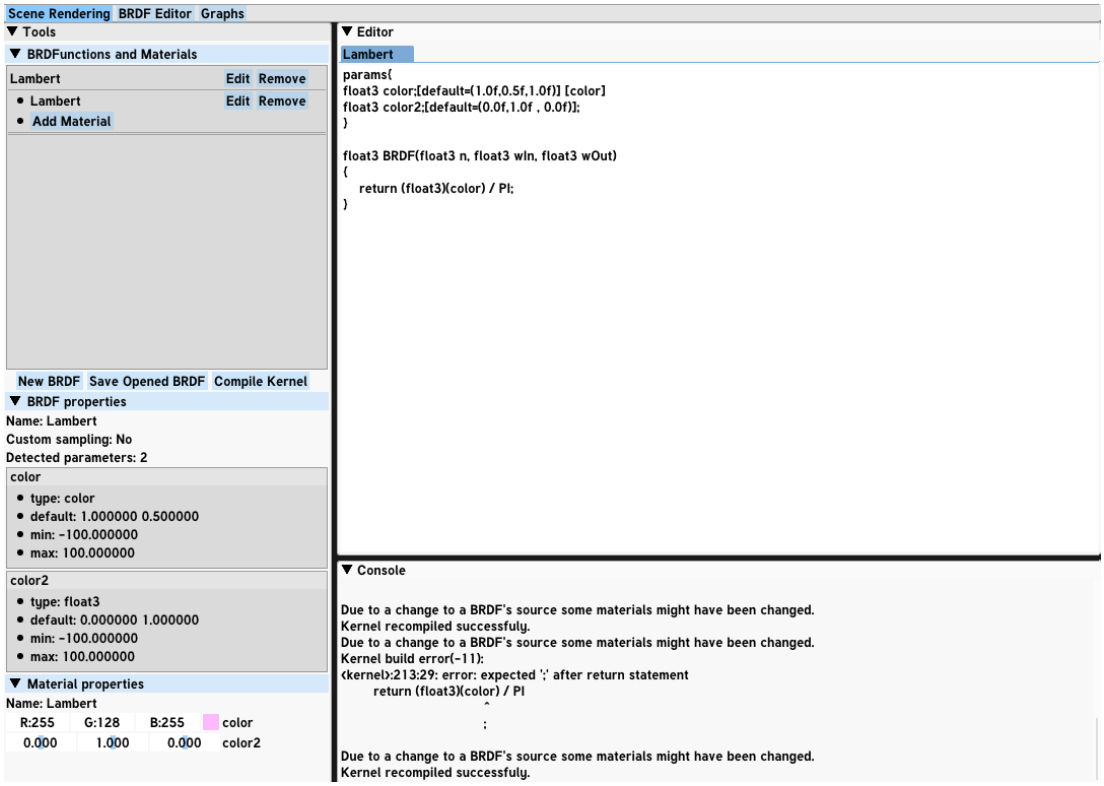

Figure 10.4: We can freely play with the BRDF, we have removed the custom sampling, added another parameter. This time without the `color` tag. See appendix A.2 for the syntax for the parameters. We have also at one point removed a semicolon, compiling this BRDF lead to a compiler error in the console. Adding it back and recompiling again solved the problem.

Next, we will create a new material called *LambertRed*, if we click on the *edit* next to it, we can change its parameters in the lower left corner, in particular we've changed the *color* parameter to red color. Now, we will move to the third part of the program - graphs. Make sure that the kernel does not contain any errors by compiling it before moving on, as the graphs would not work otherwise.

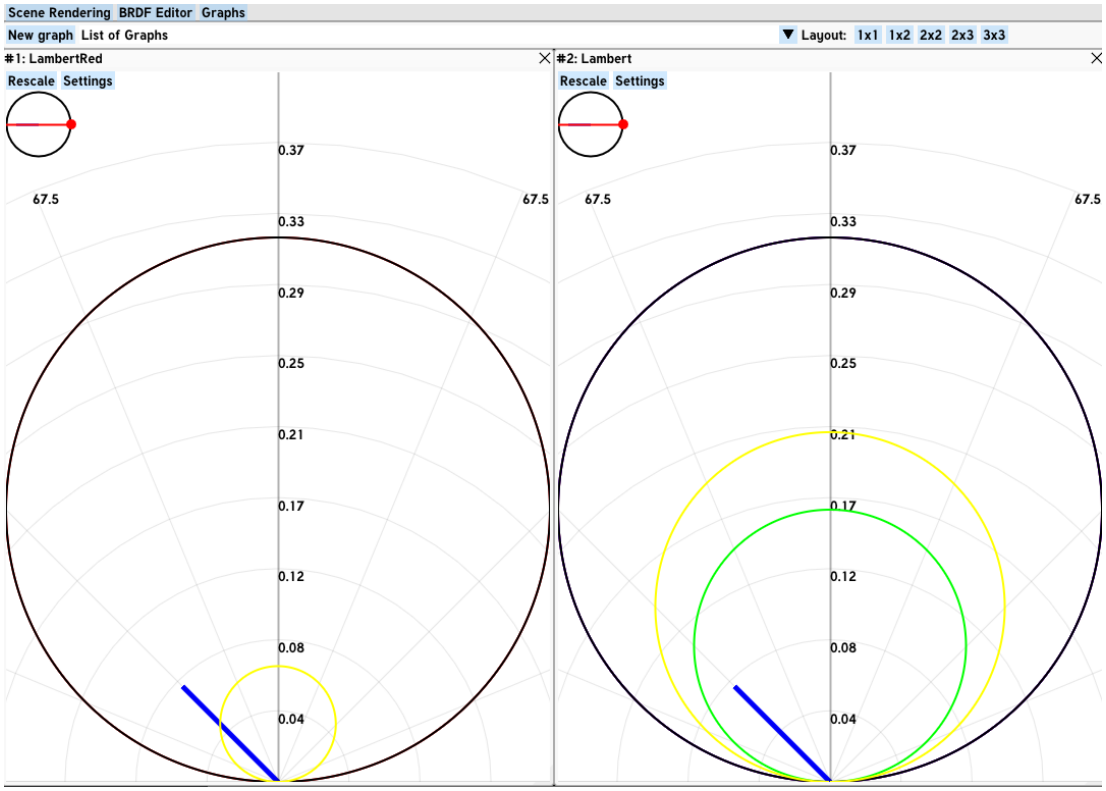

Figure 10.5: After switching to graphs we have created a graph for each of the two materials present in the scene. Their windows can be automatically arranged by the series of buttons on the top panel. Each windows shows the plotted material, more precisely for a given incoming direction(blue) the graph shows values for outgoing directions in the chosen plane (red). The circle in the top left corner represent the top-view of this interaction, using left,right mouse buttons the user can set the directions. Hovering over the graphs shows the elevation angle and the associated value of that point.

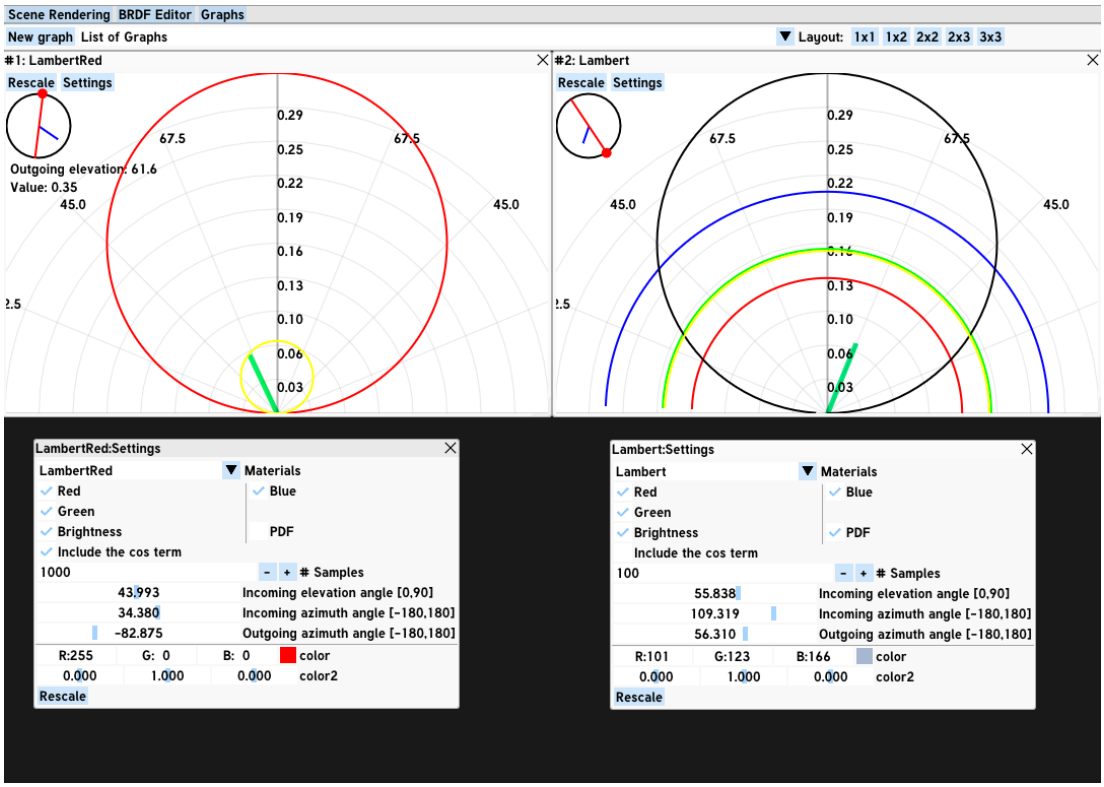

Figure 10.6: We have rearranged the kernels into 2x2 grid and opened the settings windows for both. These allow precise control over the graphs. Using checkboxes we can changed which values get plotted, including the custom sampling(PDF, black). We have changed the graphs a little to show their capabilities. The windows can be closed using the ”X” button and reopened by opening the list of graphs, this is also the place where can the graphs be deleted. The user can switch between the editor and these graphs as long as they always properly resolve any errors.

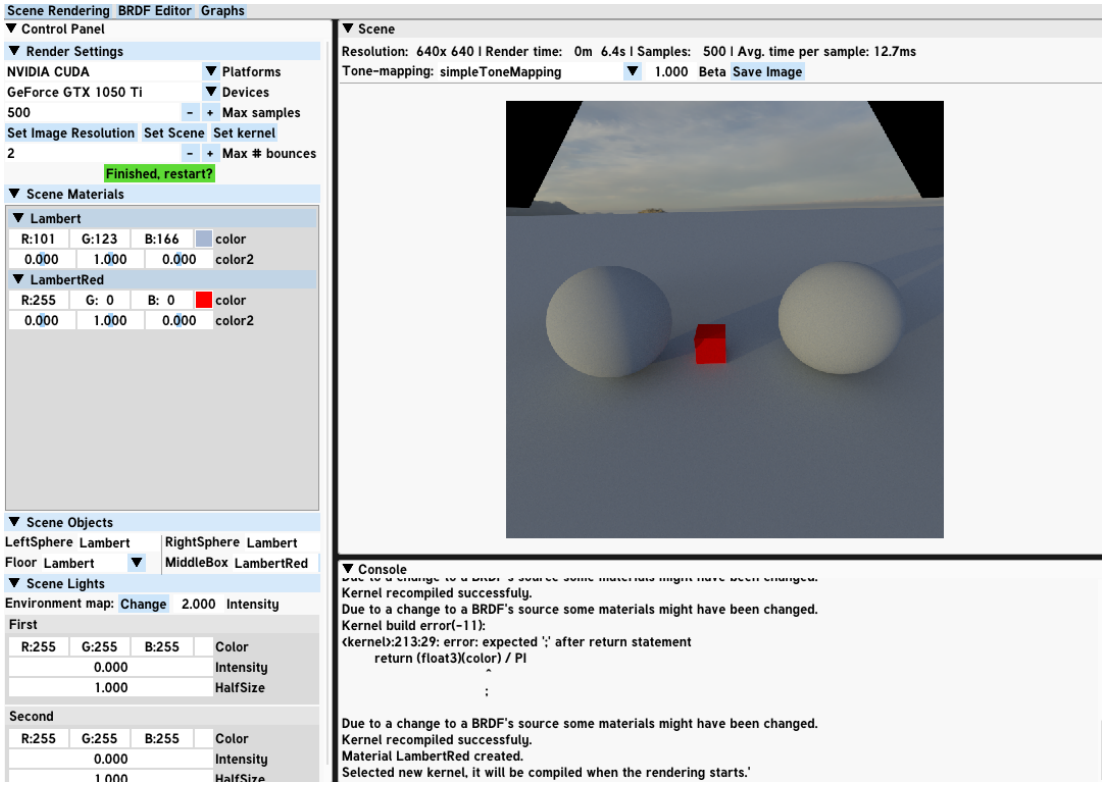

Figure 10.7: Returning back to the rendered scene. We have add a new environment map `cape_hill_2k.hdr` (loading will take very long in the debug mode), changed the algorithm to `pathTracer-DL-IS`, dimmed the area lights (the two black parts), rotated the camera around, and finally rendered the scene. Note that the time at the top panel shows the time spend executing the kernel, it does not include the execution of the rest of the program.

10.2 Writing a custom scene

The scene must be written in a `.json` file using JSON syntax, C++-style comments are allowed. Please see any examples distributed together with the program, all names are case-sensitive. The top level object must contain:

1. `camera` : Objects containing three arrays: `position`,`dirup` which correspond to 3D coordinate space for the camera. So we will refer to these 3-component arrays as vectors.

2. `BRDFs`: A list of (`name`,`filename`) pairs specifying the included BRDFs, the file name is relative to the executable.

3. `materials`: Similar to `BRDFs` but instead of a `filename` a BRDF's name must be present in `BRDF` field.

4. `lights` and `objects` array.

In addition to these, there are two optional values - `envMap` string value holding the file name for the environment map and envMapIntensity multiplying the values in the loaded map.

The possible objects in the `lights` array must have a string value named `type` and the value must be a one of `sphere, plane, cuboid, mesh`. Similary all lights must have this attribute too with allowed values being `point,area,sphere`. They all have fairly self-explanatory attributes and all of them are mandatory, see created examples.

There are three interesting arguments a `mesh`. These are `yaw,pitch,roll` representing rotation angles in degrees for individual axes X,Y,Z respectively. We use the OpenGL convention - X is right, y up, Z points away from the monitor towards the reader.

Bibliography

[1] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[2] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *SIGGRAPH*, 1994.

[3] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[4] Eric P. Lafortune. Mathematical models and monte carlo algorithms for physically based rendering. 1995.

[5] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[6] Stefan Weinzierl. *Introduction to Monte Carlo methods*, chapter 3,4. arXiv:hep-ph/0006269. 2000.

[7] tavianator. Fast, branchless ray/bounding box intersections. `https://tavianator.com/fast-branchless-raybounding-box-intersections/`. Accessed: 14.7.2019.

[8] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[9] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, SCCG '11, pages 7–12, New York, NY, USA, 2013. ACM.

[10] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), January 2007.

[11] Rosana Montes Soldado and Carlos Ureña Almagro. An overview of brdf models. 2012.

[12] Kenneth E. Torrance and Ephraim M. Sparrow. Theory for off-specular reflection from roughened surfaces*. 1967.

[13] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1:7–24, 1982.

[14] Csaba Kelemen and László Szirmay-Kalos. A microfacet based coupled specular-matte brdf model with importance sampling. 2001.

[15] Christophe Schlick. An inexpensive brdf model for physically-based rendering, 1994.

[16] Michael Ashikhmin and Peter Shirley. An anisotropic phong brdf model. *Journal of Graphics Tools*, 5:25–32, 2000.

[17] Michael Oren and Shree K. Nayar. Generalization of lambert's reflectance model. In *In SIGGRAPH 94*, pages 239–246. ACM Press, 1994.

[18] Jos van Ouwerkerk. van ouwerkerk's rewrite of the oren nayar brdf. `http://shaderjvo.blogspot.com/2011/08/van-ouwerkerks-rewrite-of-oren-nayar.html`. Accessed: 14.7.2019.

[19] Wojciech Matusik, Hanspeter Pfister, Matt Brand, and Leonard McMillan. A data-driven reflectance model. *ACM Transactions on Graphics*, 22(3):759–769, July 2003.

[20] Intel® digital random number generator (drng) software. `https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide#inpage-nav-4`. Accessed: 14.7.2019.

[21] Amd random number generator library. `https://developer.amd.com/amd-aocl/rng-library/`. Accessed: 14.7.2019.

[22] Jiří Matějka. Brdf dílna, 2012.

[23] David B. Thomas. The mwc64x random number generator. `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html`. Accessed: 14.7.2019.

[24] Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 2014.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[26] Peter Shirley and Changyaw Wang. Monte carlo techniques for direct lighting calculations to appear in acm transactions on graphics. 1994.

[27] D. Cline, A. Razdan, and P. Wonka. A comparison of tabular pdf inversion methods. *Computer Graphics Forum*, 28(1):154–160, 2009.

[28] George Marsaglia, Wai Wan Tsang, and Jingbo Wang. Fast generation of discrete random variables. *Journal of Statistical Software, Articles*, 11(3):1–11, 2004.

[29] Adrià Forés, Sumanta N. Pattanaik, Carles Bosch, and Xavier Pueyo. Brdflab: A general system for designing brdfs. In *CEIG*, 2009.

[30] NVIDIA. Nvidia cuda zone. `https://developer.nvidia.com/cuda-zone`. Accessed: 14.7.2019.

[31] Khronos Group. Opencl. `https://www.khronos.org/opencl/`. Accessed: 14.7.2019.

[32] AMD. Gpuopen. https://gpuopen.com/. Accessed: 14.7.2019.

[33] Intel. Intel opencl sdk. https://software.intel.com/en-us/opencl-sdk. Accessed: 14.7.2019.

[34] Khronos. Features of opengl versions. https://www.khronos.org/opengl/wiki/History_of_OpenGL#OpenGL_4.3_.282012.29. Accessed: 14.7.2019.

[35] Khronos Group. Sycl. https://www.khronos.org/sycl/. Accessed: 14.7.2019.

[36] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143, New York, NY, USA, 2013. ACM.

[37] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Comput. Graph. Forum*, 29:289–298, 05 2010.

[38] ITU. Bt.709 parameter values for the hdtv standards for production and international programme exchange. https://www.itu.int/rec/R-REC-BT.709-6-201506-I/en. Table 3, Accessed: 14.7.2019.

List of Figures

A. Attachments

A.1 Headers of the used kernel functions

```
__kernel void render(
const global Sphere* spheres, const global Plane* planes,
const global Cuboid* cuboids, const global BVHNode* bvhNodes,
const global Triangle* triangles, const global Mesh* meshes,
const global PointLight* pLights, const global AreaLight* aLights,
const global SphereLight* sLights, const Scene scene,
global MatInfo* matInfos, global Param* matParams,
global float4 * buffImage, global uint2* rngState,
read_only image1d_t envMarginalRowHist,
read_only image2d_t envRowHists,
read_only image2d_t envMap,
float envMapIntensity,int numSamples,int maxBounces);


float3 BRDFEval(
MatInfo matInfo, float3 normal, float3 wIn, float3 wOut,
__global Param* paramBuffer);
float3 BRDFSampleEval(MatInfo matInfo, float3 normal, float3 wIn,
float3* wOut, RNGState* rngState, float* pdf,
__global Param* paramBuffer);


float GetwInPDF(MatInfo matInfo, float3 n, float3 wOut,
float3 wIn, __global Param* paramBuffer);
```

A.2 BRDF example

The following code shows an example of valid BRDF definition, including all possible parameter types and optional tags.

```
params{//Must be on the first line
float3 x;[default=(1.0f,0.5f,1.0f)] [color]
float2 y;[min=1.0f][max=2.0f][default=(0.0f,1.0f)]
int i;[step=0.1f]
}
//[color] tag will treat the value as color, enables color picker
    in UI.
//[step],[min],[max] contain always one float value
//[step] controls the draggin speed of UI sliders.

//Predefined constants available in every BRDF.
#define PI 3.1415926535f
#define TWOPI (2.0f*PI)
#define HALFPI (PI/2.0f)
#define EPSILON 0.000001f

float3 BRDF(float3 n, float3 wIn, float3 wOut)
{
//Defined parameters can be used in any of these three functions.
 return (float3)(color) / PI;
}
float3 SampleBRDF(float3 n, float3 wIn, float3* wOut, float* pdf,
    RNGState* rngState)
{
float tn1,tn2;
//The two predefined functions
// wIn is currently unused by the function and can be anything.
*wOut=genCosineDir(wIn,n,rngState,pdf);
//Generates orthonormal base around the normal.
genBasis(n,&tn1,&tn2);

return (float3)(color) / PI;
}

float GetwInPDF(float3 normal, float3 wOut, float3 wIn){
return dot(normal,wIn)/PI;
}
```